Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня магістра

студента Морозова Богдана Денисовича
(ПІБ)

академічної групи 123м-20-1
(шифр)

спеціальності 123 Комп'ютерна інженерія
(код і назва спеціальності)

за освітньо-професійною програмою 123 Комп'ютерна інженерія
(офіційна назва)

на тему «Дослідження ефективності використання технологій контейнеризації в комп'ютерних системах»
(назва за наказом ректора)

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	проф. Гнатушенко В.В.			
розділів:				
дослідження контейнеризації	проф. Гнатушенко В.В.			
Розробка програмного забезпечення	проф. Гнатушенко В.В.			

Рецензент				

Нормоконтролер	проф. Цвіркун Л.І.			

Дніпро
2022

ЗАТВЕРДЖЕНО:
завідувач кафедри
інформаційних технологій та
комп'ютерної інженерії
(повна назва)
_____ проф. Гнатушенко В.В.
(підпис) (прізвище, ініціали)

ЗАВДАННЯ
на кваліфікаційну роботу ступеня магістра

студента Морозова Б.Д. **академічної групи** 123м-20-1
(прізвище та ініціали) (шифр)
спеціальності 123 «Комп'ютерна інженерія»

за освітньо-професійною програмою 123 Комп'ютерна інженерія
(офіційна назва)
на тему «Дослідження ефективності використання технологій контейнеризації в комп'ютерних системах»
затверджену наказом ректора НТУ «Дніпровська політехніка» від _____ №____

Розділ	Зміст	Термін виконання
Стан питання та постановка завдання	На основі матеріалів практик, інших науково-технічних джерел сформулювати завдання, конкретизувати предмет та мету роботи	10.11.2021
Технічні вимоги до системи	На основі матеріалів практик, інших науково-технічних джерел сформулювати технічні вимоги до системи	29.11.2021
Спеціальна частина	Розв'язати завдання з розробки інфраструктури розгортки для комп'ютерної системи з опрацюванням контейнеризації та оркеструванням контейнерів	15.12.2021

Завдання видано _____ проф. Гнатушенко В.В
(підпис керівника) (прізвище, ініціали)
Дата видачі _____
Дата подання до екзаменаційної комісії _____
Прийнято до виконання _____ Морозов Б.Д.
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 62 с., 33 рис., 1 табл., 1 додаток, 13 джерел.

Об’єкт розробки: розподілена комп’ютерна система управління кадрами для дизайнерської студії с з використанням мікросервісної так мікрофронтенд архітектури.

Мета: дослідити ефективність контейнеризації мікросервісних фронтенд застосунків.

Розподіл монолітної архітектури фронтенд застосунку на мікросервіси має на меті значне поліпшення процесу підтримки та на розгортання системи на віддалених ресурсах.

Таке вдосконалення обумовлене необхідністю застосування технологій постійної інтеграції та постійного розгортання для підвищення ефективності і швидкості розробки програмного продукту шляхом скорочення часу на обслуговування системи.

Розробка мікрофронтенд архітектури виконана відповідно до завдання на дипломну роботу магістра.

Розроблена схема архітекртури мікросервісів реалізована у вигляді веб-застосунку і протестована згідно вимог підприємства.

Результати розподілу системи у вигляді графіків, таблиць, наводяться і описані у пояснювальній записці та додатках.

CONTAINER, DOCKER, FRONT-END, KUBERNETES, DEVOPS

ЗМІСТ

Перелік умовних позначень, скорочень і термінів 6
Вступ 8
1 Стан питання та постановка завдання 9
1.1 Обґрунтування необхідності використання мікросервісної архітектури 9
1.2 Аналіз інструментів які будуть використані 13
1.3 Переваги використання віртуалізації 17
2 Вимоги до архітектури системи 21
2.1 Вимоги до системи 21
2.1.1 Нефункціональні вимоги до системи 21
2.1.2 Вимоги до програмної частини 21
2.1.3 Вимоги до архітектури системи 22
2.1.4 Вимоги до надійності інформаційної системи 27
2.1.5 Вимоги щодо безпеки 27
2.2 Вимоги до розгортання 28
2.2.1 Постійна розгортка та постійна інтеграція 28
2.2.2 Централізовані обчислювальні платформи 29
3 Розробка програмного забезпечення 31
3.1 Побудова мікросервісів 31
3.2 Шаблон міграції на мікросервіси 32
3.3 Робота з образами 34
3.3.1 Створення образів 34
3.3.2 Реєстр образів 36
3.4 Теорія роботи контейнерів 37
3.4.1 Запуск контейнерів 37

3.4.2 Кластери контейнерів 39
3.5 Збереження даних 39
3.6 Розгортка застосунку в мережі 40
3.7 Забезпечення мережевого доступу до додатку 42
3.8 Створення кластера Kubernetes 47
3.9 Налаштування автоматичного масштабування 55
3.10 Розгортання контейнерного додатку 59
Висновки 61
Список використаної літератури 62
Додаток А ТЕКСТ ПРОГРАММИ 63

ПЕРЕЛІК УМНОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Kubernetes — це система контейнерного оркестрування з відкритим кодом для автоматизації розгортання, масштабування та керування комп'ютером. Його ціль — надати «платформу для розгортання, масштабування та автоматизації». Він працює з рядом інструментів для контейнерів і запускає контейнери в кластері, часто з зображеннями, створеними Docker.

DOCKER — це набір продукт (PaaS), який використовує віртуалізацію на рівні ОС для розгортки програмного забезпечення на середовищах, які називаються контейнерами. Контейнери ізольовані один від одного і поєднують бібліотеки, програмне забезпечення, та конфігураційні файли; вони можуть спілкуватися між собою через заздалегідь визначені канали. Оскільки контейнери використовують однакові служби в системному ядрі, вони використовують значно менше ресурсів, ніж віртуальні машини.

UI (User's Interface) —інтерфейс користувача програмного забезпечення

API (англ. application programming interface) — це частина сервера, яка приймає запити та надсилає відповіді.

AJAX (Asynchronous Javascript and XML) — асинхронний JavaScript і XML, концепція створення інтерфейсів користувача для веб-додатків.

Скрін (від англ. Screen - екран) — в значенні одиниці додатку зазвичай використовується як термін для всього що бачить користувач в дану одиницю часу.

SPA (Single page application) — веб-додаток, який використовує AJAX, щоб уникнути необхідності запитувати інформацію про веб-сторінку від сервера під час зміни інтерфейсу.

MVC (Model View Controller) — це шаблон проектування програмного забезпечення, який зазвичай використовується для розробки користувацьких інтерфейсів.

AWS (Amazon Web Services) — популярна платформа для хмарних обчислень за запитом.

DOM (Document Object Model) — це специфікація що описує можливі шляхи роботи в документом для розробника або просто об'єктна модель документа.

DevOps (акронім від англ. development і operations) —це комбінація розробників програмного забезпечення (dev) і операцій (ops). Його визначають як методологію розробки програмного забезпечення, яка має на меті інтегрувати роботу команд з розробки програмного забезпечення та операцій з програмним забезпеченням шляхом сприяння розвитку культури співпраці та спільної відповідальності.

ВСТУП

Із розвитком мережі інтернет та інформаційних технологій із неймовірною швидкістю зростають вимоги до комп'ютерних систем, складність їх розробки та підтримки. Все частіше з'являються нові практики та підходи до розробки та розгортання системи які могли б відповідати сучасним потребам підприємств.

Мікросервісна архітектура та контейнеризація є загальноприйнятим сучасним рішенням проблеми обслуговування та розгортки комп'ютерних систем. Ці технології спрямовані допомогти пришвидшити процеси розгортки які починаються коли безпосередньо сам продукт вже готовий, але треба переконатися в тому що продукт буде своєчасно оновлено та правильно розгорнуто в мережі де до нього матимуть доступ користувачі.

Основними сучасними інструментами для задоволення таких потреб підприємств є Kubernets та Docker, ці два інструменти є частиною імплементації DevOps практик. З часом та беручи до уваги нові стандарти ці інструменти зазнали значного розвитку, та продовжують поліпшуватися, часто з'являються нові підходи до їх використання.

В рамках цією роботи програмний застосунок буде поділено на логічні частини – мікрофронтенди. Поділені частини застосунку буде контейнеризовано за допомогою Docker. Отримані образи докер будуть орекстровані за допомогою Kubernetes.

1 СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Обґрунтування необхідності використання мікросервісної архітектури

Компанія, яка замовила реорганізацію продукту, є невеликою компанією, що займається професійним розробкою дизайну і брендингом для клієнтів по всьому світу. У підприємства не має власного сервера та жодних мережевих пристроїв в користуванні, оскільки це економічно невигідно, тому вони використовують хмарні технології, переважно від Amazon та Google. Платформа AWS Amazon використовується для зберігання системних ресурсів.

На разі з постійним зростанням системи топ-менеджментом прийнято рішення замовити консалтингові послуги в галузі інформаційних технологій з метою поліпшити наявну інфраструктуру там зробити можливим подальше розширення програмного продукту. На рисунку 1.2 наведена поточна повна архітектурна схема системи.

Під час аудиту було виявлено ряд недоліків:

- Процеси розробки не дозволяють великій команді ефективно працювати над єдиним продуктом
- Розгортка системи виконується вручну. Оновлення або виправлення будь якої частини застосунку призводить до необхідності розгорнути всю систему заново, така маніпуляція займає від двох до п’яти робочих днів та потребує залучення декількох спеціалістів
- Не налаштовані системи постійної інтеграції та постійного розгортання, це дозволяє неякісному коду потрапити напряму до користувачів

- Нові розробники витрачають тижні на локальну розгортку проекту, та потребують допомоги та нагляду від колег, навіть для того щоб виконати базовий запуск

Архітектура, побудована в цій роботі, покликана частково або повністю вирішити ці моменти.

Проблема дистанційного сканування буде вирішена за допомогою систем постійної інтеграції та постійного сканування. Завдяки інтеграції цих механізмів можна буде автоматизувати розгортання програми, позбутися від необхідності залучення фахівців на цьому етапі та скоротити весь процес до 30 хвилин.

Проблему неякісного коду в системі можна вирішити шляхом налаштування статичних аналізаторів і автоматичного тестування коду на етапі безперервної інтеграції. Вам також потрібно налаштувати систему перегляду коду на етапі інтеграції.

Проблему сканування як на локальних машинах, так і на віддалених можна вирішити шляхом контейнерування всіх сервісів у системі. Цього можна досягти за допомогою Docker і Kubernetes. Docker дозволить вам створити образ системи з уже встановленим сервісом, щоб усунути необхідність будь-яких додаткових кроків під час сканування. Kubernetes, у свою чергу, допоможе вам керувати кількома існуючими образами за допомогою різних сервісів, які становлять комп’ютерну систему. На рисунку 1.1 зображена запропонована зміна в архітектурі комп’ютерної системи.

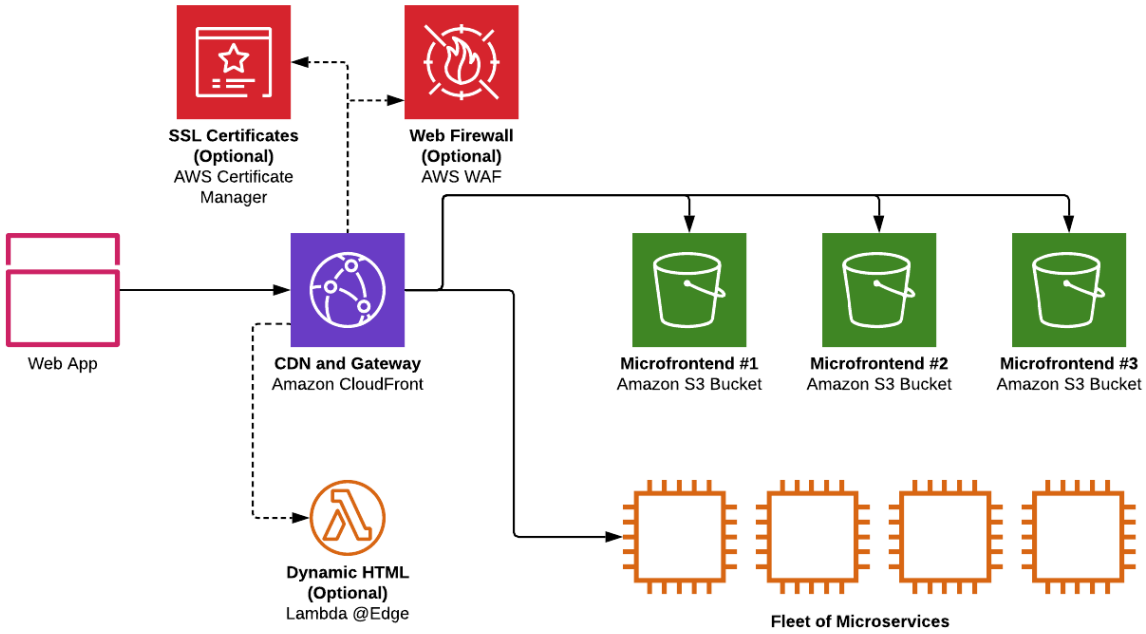

Рисунок 1.1 – Запропонована архітектура

Реалізація мікросервісної архітектури, системи постійної розгортки та інтеграції з використанням технологій контейнеризації є необхідним для подальшого функціонування продукту підприємства та основним напрямком цієї дипломної роботи.

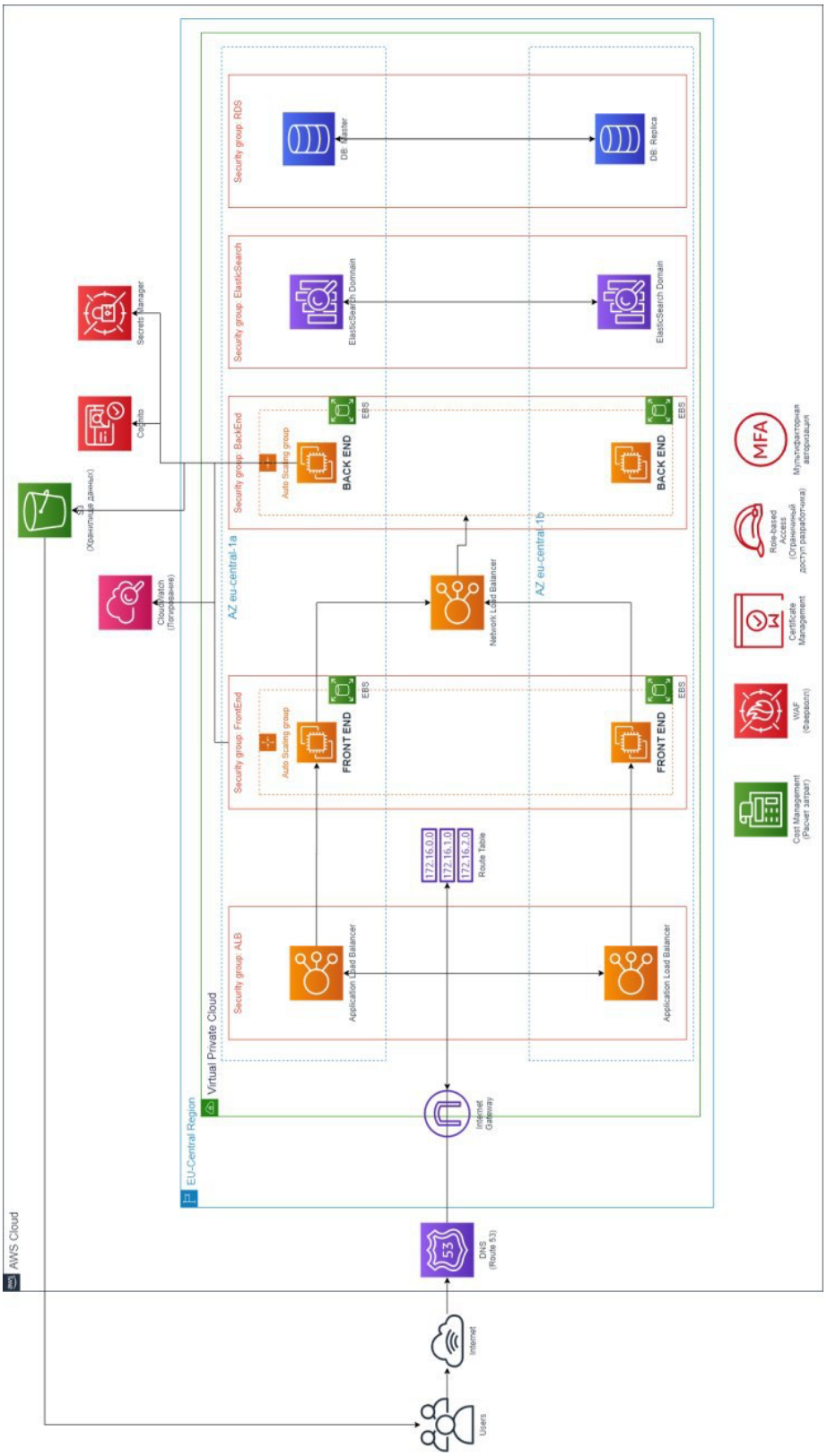

Рисунок 1.2 – Архітектурна схема системи

1.2 Аналіз інструментів які будуть використані

Docker – це програмне забезпечення з відкритим кодом, найпопулярніша платформа для управління контейнерами. Щоб зрозуміти принципи роботи Docker, його часто порівнюють з роботою транспортних перевезень. Колись транспортні компанії стикалися з проблемою перевезення різних типів товарів. Як одна вантажівка може перевезти товари різних розмірів і типів?

Із введенням контейнерів це стало можливим. Вантаж різного розміру розподілений по стандартизованих контейнерах, які перевозяться одним і тим же транспортним засобом.

Під час розробки програми вам потрібно надати код разом з усіма його компонентами як сервер, бази даних, бібліотеки тощо. Ви можете опинитися в ситуації, коли програма працює на вашому апаратному забепеченні, але відмовляється включатися і виконовати свої функції на пристрої замовника або іншого користувача. Це вирішується шляхом створення програмної незалежності від системи. Docker розділяє ресурси ядра ОС на контейнери Docker, які працюють як окремі процеси. Він вирішує багато проблем, пов’язаних зі створенням контейнерів, розміщенням в них додатків, управлінням процесом, а також тестуванням програмного забезпечення та його окремих компонентів.

Docker в цьому випадку, необхідний для ефективного використання ресурсів та системи, розгортання програм, а також для автомасштабування та передачі в інші середовища з гарантованою стабільністю. На рисунку 1.3 показана приблизна схема компонентів у Docker.

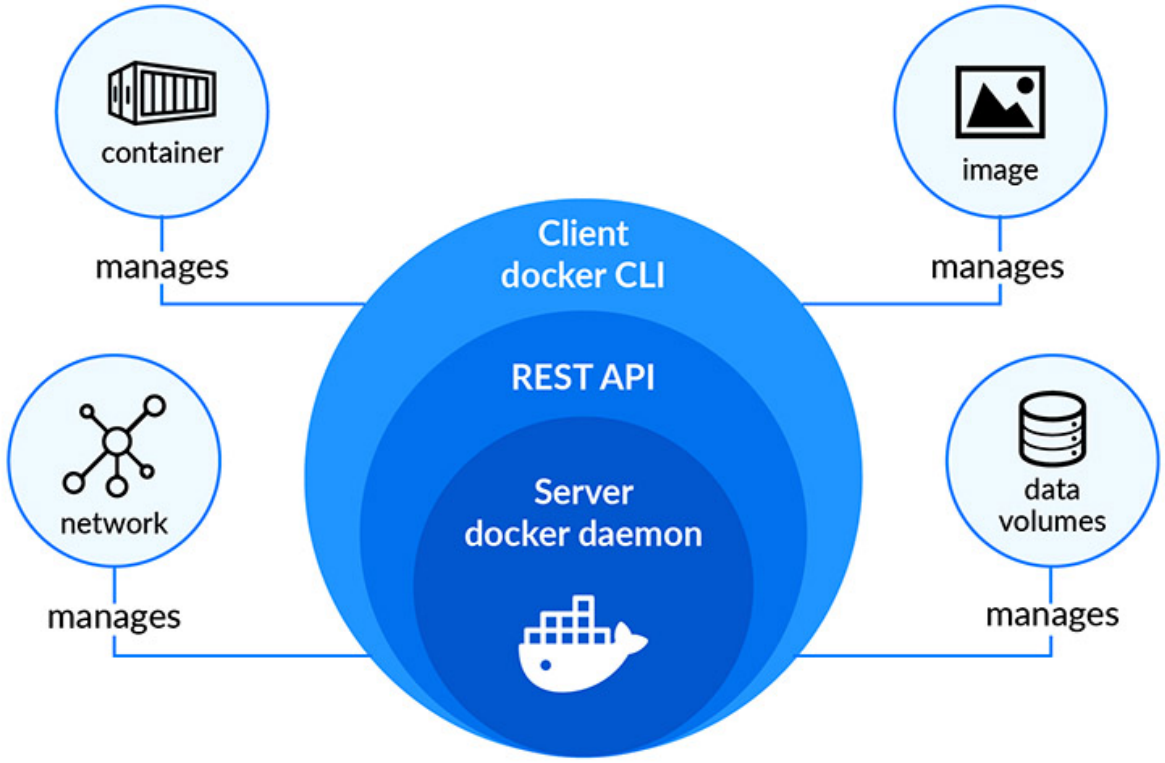

Рисунок 1.3 – Схема компонентів в Docker

Docker допомагає:

– мінімально використовувати системні ресурси
– зручно приховати або видалити фонові процеси
– просто масштабувати
– значно зменшити час між розробкою і виконанням коду
– швидше тестувати
– швидко розгортати
– швидше створювати додатки

Docker робить це за допомогою легкої платформи віртуалізації контейнерів. По суті, docker дозволяє запускати практично будь-яку програму,

яка буде точно ізольована в контейнері. Безпечна ізоляція надає змогу запускати безліч контейнерів лише на одному хості.

Docker характеризується досить простим синтаксисом. Тому для досвідчених IT-фахівців і новачків це досить просто. Програмне забезпечення сумісне з усіма версіями операційних систем Linux і Windows, тому область застосування Docker практично необмежена.

Kubernetes — це інструмент для управління складними робочими навантаженнями, що працюють на великій кількості контейнерів, розподілених між фізичними або віртуальними хостами, і використовує механізми контейнерів як будівельний блок. Філософія Kubernetes виступає за використання контейнерів для організації програмного забезпечення для невеликих взаємопов'язаних служб або мікросервісів та автоматизованого за допомогою декларативних методів. Ця філософія, поряд з такими підходами до розробки, як Devops і безперервне розгортання, є важливими компонентами руху розвитку хмари. Так, Google і Linux Foundation створили Cloud Native Computing Foundation (CNCF) у 2015 році, який просуває власні підходи до хмарних технологій та визначає екосистему для її підтримки.

Kubernetes використовує робочу парадигму для визначення кластера або групи серверів (фізичних або віртуальних). Кожен працівник кластера запускає контейнерний механізм, і майстер Kubernetes наказує їм запускати та налаштовувати робочі навантаження. Як правило, користувач, який розгортає програми в Kubernetes, розглядає кластер як недиференційований ресурс обчислень та зберігання або як один великий комп'ютер. Додаток у Kubernetes, як правило, складається з багатьох контейнерів, організованих у служби, які споживають один одного. Контейнери створюються за допомогою инструмента Docker. Ця архітектура підтримує паралельність випуску програмного забезпечення, дозволяючи швидке та відносно низький ризик розгортання виправлень помилок, оновлення та випуску функцій.

Після того, як програмна система знаходиться в Kubernetes, площина керування платформою активно керує розгортанням за допомогою планувальника, який керує різними співробітниками. Дескриптори програми, розгорнуті в Kubernetes, представляють декларативний «бажаний стан» програми, який планувальник виявляє, розгортає та підтримує у вигляді автоматизованого замкнутого циклу. На рисунку 1.4 зображено схему компонентів в Kubernetes.

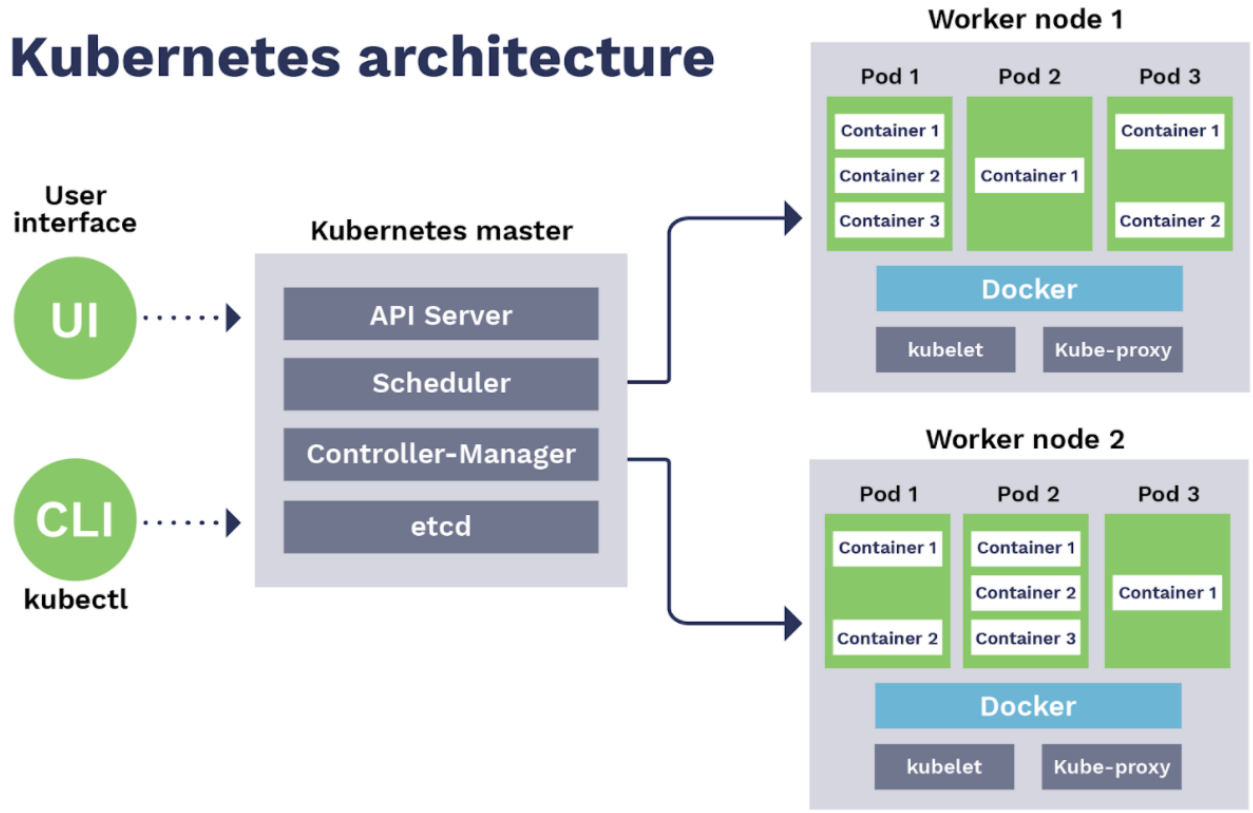

Рисунок 1.4 – Схема компонентів в Kubernetes

1.3 Переваги використання віртуалізації

Існує сім типів різної віртуалізації, які відрізняються залежно від апаратного забезпечення та програм, що використовуються. Нижче ми розглянемо три типи віртуалізації, які належать до підрозділу віртуалізації операційних систем.

Віртуалізація пропонує значні переваги практично для будь-якого бізнесу або середовища розробки. Це стало основною стратегією підвищення ефективності IT.

Давайте подивимося, як віртуалізація може заощадити гроші, оптимізувати DevOps і підвищити доступність ваших послуг.

Знижені витрати. Обчислювальна потужність має свою ціну. Якщо єдиний спосіб отримати більше ресурсів — це придбати нове обладнання, ціна стає величезною. За допомогою тактики віртуалізації ви можете уважно оглянути наявну інфраструктуру та визначити витрачені або неактивні обчислювальні ресурси.

Занадто часто організації розгортають сервери для запуску програм, які споживають лише частину доступних ресурсів. Такі сервери ніколи не використовують весь свій потенціал. Що ще гірше, коли їхні програми не запущені, ці сервери повністю неактивні.

У віртуалізованому середовищі ви можете призначити кожній віртуальній машині точну кількість обчислювальної потужності, необхідну їй для виконання своєї роботи. Решта ресурси потім доступні для інших віртуальних машин та їхніх програм.

Витрати на віртуалізацію майже завжди нижчі, ніж вартість придбання та обслуговування додаткового обладнання.

Стійкість. Віртуалізоване серверне середовище не прив’язане до обладнання, як традиційне середовище. Ви можете легко створювати резервні копії, копіювати та клонувати віртуальні машини на різне фізичне обладнання.

Очікування, коли нове обладнання буде готове до розгортання, може зайняти дні, тижні чи навіть місяці. Тим часом ви можете розгорнути резервну копію ВМ за лічені хвилини. Коли закон Мерфі нарешті наздожене вас, ви будете вдячні, що зможете швидко розгорнути свою віртуальну машину на іншій машині, в іншому місці, з мінімальними клопотами.

Висока доступність. Оскільки ви можете клонувати віртуальну машину майже без зусиль, ви можете легко налаштувати надлишкові віртуалізовані середовища з надзвичайно високою доступністю. Завдяки автоматичному моніторингу стану віртуальної машини та швидкому перемиканню на резервну віртуальну машину в разі збою, віртуалізація забезпечує надзвичайно надійну систему без жодної точки збою в апаратному чи програмному забезпеченні.

Ці системи «відмови» дозволяють безперешкодно продовжувати роботу вашої віртуальної машини з її останнього робочого стану. Це максимізує доступність послуг незалежно від того, що піде не так.

Ви можете віддалено контролювати, налаштовувати та перезапускати все своє віртуальне середовище. Це надає розробникам постійний доступ, незалежно від того, наскільки вони віддалені від фізичного обладнання, що допомагає ще більше пом’якшити будь-які потенційні простої.

Підвищена ефективність. Віртуальне середовище набагато легше підтримувати, ніж фізичне. Замість того, щоб керувати численними фізичними серверами, які потребують індивідуальної уваги, віртуалізація дає змогу налаштовувати, відстежувати й оновлювати всі ваші віртуальні машини з однієї машини. Це заощаджує час на розгортання оновлень, впровадження виправлень безпеки та встановлення нового програмного забезпечення.

З меншою кількістю фізичного обладнання, про яке потрібно турбуватися, ваш IT-відділ витрачає менше часу на обслуговування фізичних машин. Ваші розробники насолоджуються ефективністю швидкого розгортання віртуальної машини, не турбуючись про додавання нового обладнання.

Віртуальні середовища за своєю природою є масштабованими. Ви можете легко розгорнути багато екземплярів однієї віртуальної машини, щоб справлятися з великим навантаженням, пропонуючи ефективну масштабованість, яка завжди готова підтримувати та підтримувати зростання.

Легкий DevOps. Віртуалізація – найкращий друг розробника. Він ефективно сегментує середовище виробництва та розробки без додаткового обладнання.

Клонувати віртуальну машину для налаштування середовища тестування неймовірно просто. Там ви можете тестувати функції та помилки сквоша, не впливаючи на ваш живий продукт.

У традиційних апаратних середовищах розробникам доводиться турбуватися про всі оновлення та обслуговування своїх машин розробки. Підтримка точного представлення живих серверів для тестування також є постійною проблемою.

ВМ швидко вирішують усі ці проблеми. Віртуалізація забезпечує доступ на вимогу до нескінченної кількості ідеально реплікованих віртуальних машин для розробників.

Розробники використовують переваги віртуалізації, щоб допомогти їм пришвидшити оновлення, покращити безпеку програмного забезпечення та підтримувати ефективний канал між розробкою, тестуванням та розгортанням.

Екологічно чисте IT. У довгостроковій перспективі віртуалізація — це екологічно чистий підхід до IT. Зменшення вимог до обладнання також

зменшує споживання електроенергії, в кінцевому підсумку мінімізуючи наш вуглецевий слід.

Це добре як для навколишнього середовища, так і для вашого результату. Економія енергії робить обслуговування серверів і центрів обробки даних дешевшим. Потім ви можете інвестувати всі ці гроші в інші підприємства, наприклад, зробити свій бізнес ще більш екологічним.

2 ВИМОГИ ДО АРХІТЕКТУРИ СИСТЕМИ

2.1 Вимоги до системи

2.1.1 Нефункціональні вимоги до системи в цілому

З боку замовника до системи було висунуто ряд вимог загальних вимог:

- Система має бути реалізована як розподілений веб-додаток та вписана в архітектуру AWS
- Безперешкодний доступ до частин або всієї зазначеної системи має відбуватися вільно через мережу інтернет (uptime ~99%)
- Системні накладні витрати не мають бути більше 1500 доларів США на рік у кожен рік що система використовується. Технічне обслуговування архітектури має бути автоматизованим і безкоштовним, або бути включеним до суми вищезазначених витрат
- Середня швидкість завантаження інтерфейсу в оновленій системі не має перевищувати поточну середньорічну швидкість завантаження
- Система повинна мати змогу визначати сервіси що вийшли з ладу та автоматично перезавантажувати їх
- Система повинна мати механізм звітування про неможливість перезавантаження для своєчасного реагування команди підтримки додатку або команди DevOps

2.1.2 Вимоги до програмної частини

З боку компанії до комп’ютерної системи було висунуто вимоги, а саме до програмної частини:

- Поточна система не повинна зазнати будь яких змін з точки зори користувача, всі компоненти системи повинні бути збережені в поточному вигляді та з наявним функціоналом
- Система протестована тестами розробника за наданими компанією сценаріями
- Систему необхідно розширити як з точки зору збільшення кількості одночасних користувачів, так і робочого навантаження в кілька разів, і мати можливість плавно уточнювати окремі елементи в майбутньому
- Система повинна й надалі залишатися SPA застосунком побудованим за допомогою фреймворку React, проте компоненти повинні бути поділені на складові частини та винесені в окремі мікрофронтенди. Компоненти мають використовувати для комунікації протокол HTTP.

2.1.3 Вимоги до архітектури системи

Архітектура системи повинна відповідати загальноприйнятим технологіям, стандартам, рекомендаціям, специфікаціям, інструментам розробки та мовам програмування. Для цієї системи обрана модель архітектури клієнт-сервер.

Технологія клієнт-сервер — це архітектура програмного забезпечення, в якій прикладна програма поділена на два логічно різні компоненти (клієнт і сервер), які взаємодіють за схемою «запит-відповідь» і вирішують свої конкретні завдання.

Архітектура сервера повинна бути реалізована через мікросервіси. Це повинно дозволити розробити серверну частину з використанням різних технологій і полегшити горизонтальне масштабування проекту.

Мікросервіс — це архітектурний стиль, у якому одна програма побудована як набір невеликих служб, кожна невелика служба працює у

своєму власному процесі та використовує простий і швидкий протокол даних (зазвичай HTTP) для зв'язку з іншими службами зв'язку. Ці послуги розроблені відповідно до вимог бізнесу та впроваджуються незалежно, використовуючи типове повністю автоматизоване середовище. Централізоване управління цими службами абсолютно найнижче. Зі боку розробника це означає можливе написання з використанням будь-яких мов програмування та технологій зберігання інформації. Порівняно із сервісно-орієнтованою архітектурою, архітектура мікросервісу полегшує реалізацію безперервного процесу доставки програмних продуктів. Мікросервіс має на меті створити єдиний додаток, тоді як сервісно-орієнтована система - це група програм, що взаємодіють між собою.

Загальний вид архітектури мікросервісної архітектури представлений на рисунку 2.1.

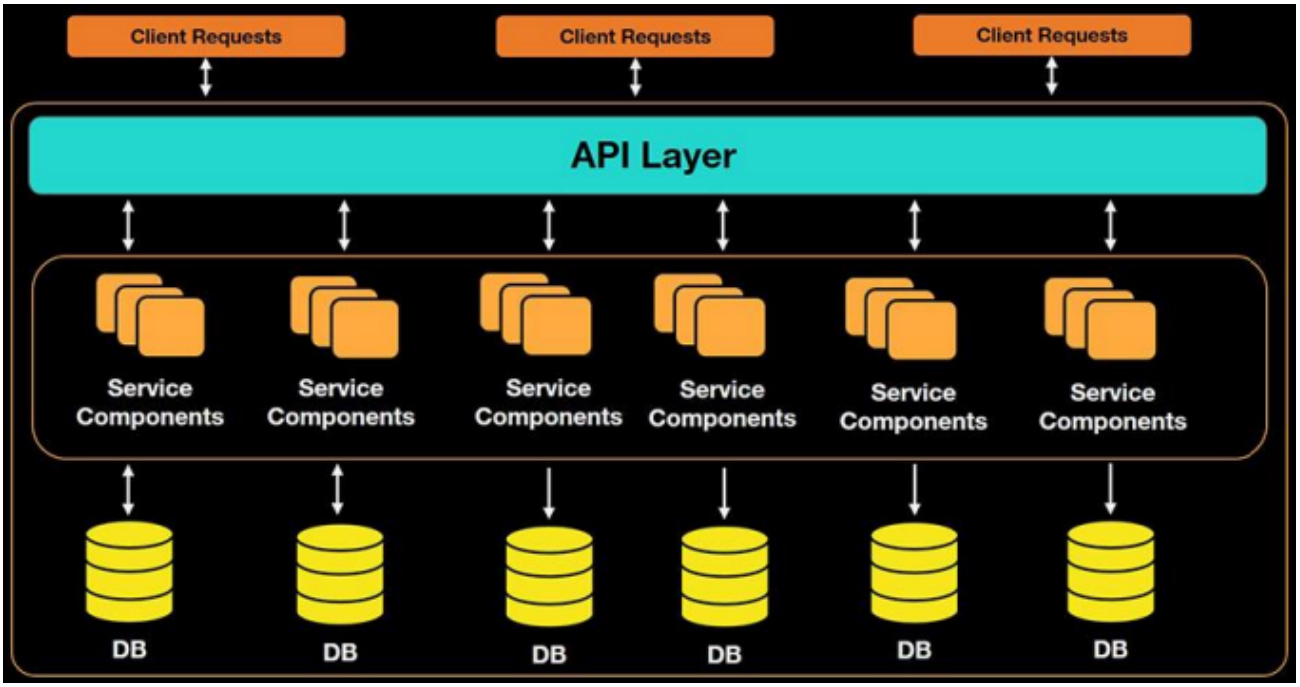

Рисунок 2.1 – Загальна схема мікросервісної архітектури

Велика частина сучасних веб-додатків реалізовані як моноліт. Монолітна архітектура — це підхід, при якому вся бізнес логіка програмного забезпечення організована та доступна в одному фізичному місці. Монолітне забезпечення має в своєму складі з шарове поєднання незв’язаних компонентів системи в один. Тобто всі команди працюють над одним репозиторієм проекту. Зміна будь-якої частини проекту зазвичай впливає на інші. Для порівняння на рисунку 2.2 показана типова структура монолітного проекту.

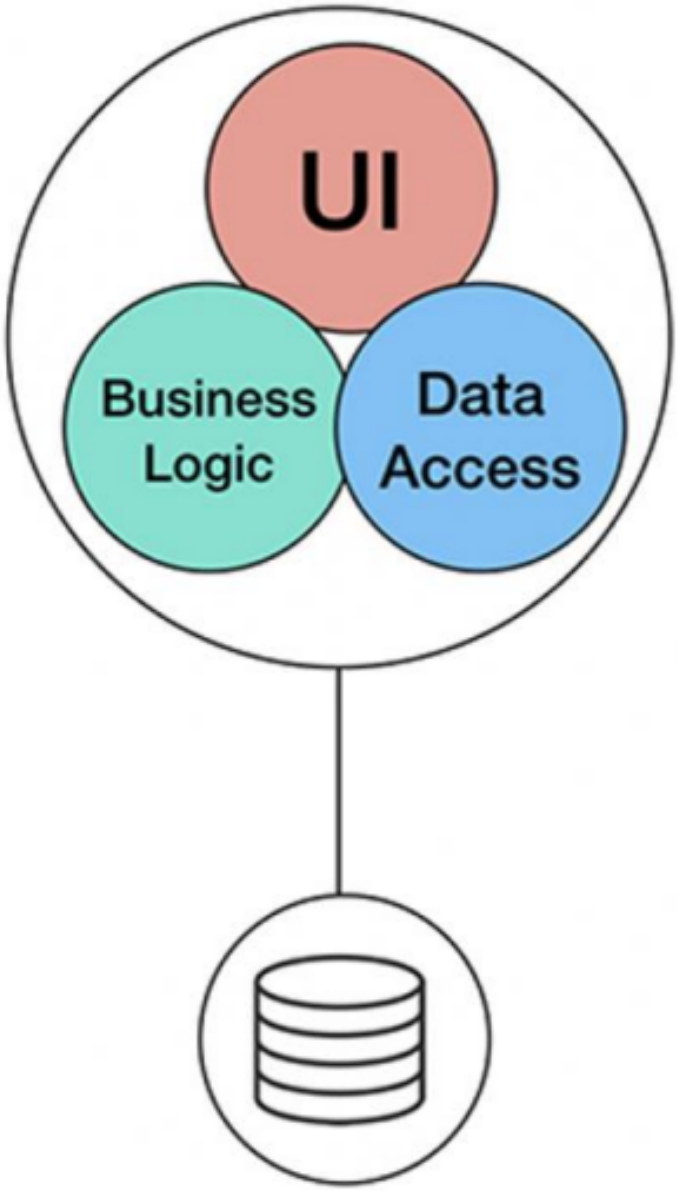

Рисунок 2.2 – Типова структура монолітного проекту

Клієнтська частина повинна бути реалізована використовуючи підхід мікрофронтенд архітектури. Цей підхід є імплементацією серверної мікросервісної архітектури для великих клієнтських додатків.

Клієнтський мікросервіс має наступні характеристики:

– повністю ізольована частина користувацького інтерфейсу, жодним чином не залежить від інших; радикальна ізольованість; буквально розробляється як окремий додаток
– - кожен клієнтський мікросервіс відповідає за певний набір бізнес-функцій від початку до кінця, тобто є повністю функціональним сам по собі
– може бути написаний на будь-яких технологіях.

Такий архітектурний підхід до розробки архітектури клієнт-сервер ефективно підходить для розробки великих довгострокових проектів, які працюють у великій кількості команд.

Оскільки монолітна система непридатна для великих проектів через міцне кріплення її компонентів, ця система використовує архітектуру мікрофронтенда.

Microfrontend — це архітектурний підхід, підхід до розробки мікросервісів та інтерфейсної веб-розробки. Нинішня тенденція полягає в створенні потужного та багатого веб-додатку, що знаходиться на вершині архітектури мікросервісів

Головні переваги мікрофронтенд-підходу:

– незалежність, різні частини застосунку - це повністю незалежні програмні додатки
– швидкість тестування, завдяки ізоляції тестування значно спрощуєтеся бо немає необхідності тестувати інші частини застосунку що оформлені як окремі програми

– паралельна розгортка, кожен компонент системи розгортається незалежно від інших що дає йому змогу не впивати та не порапляти під вплив інших компонентів

Загальна структура мікрофронтед архітектури представлена на рисунку 2.3.

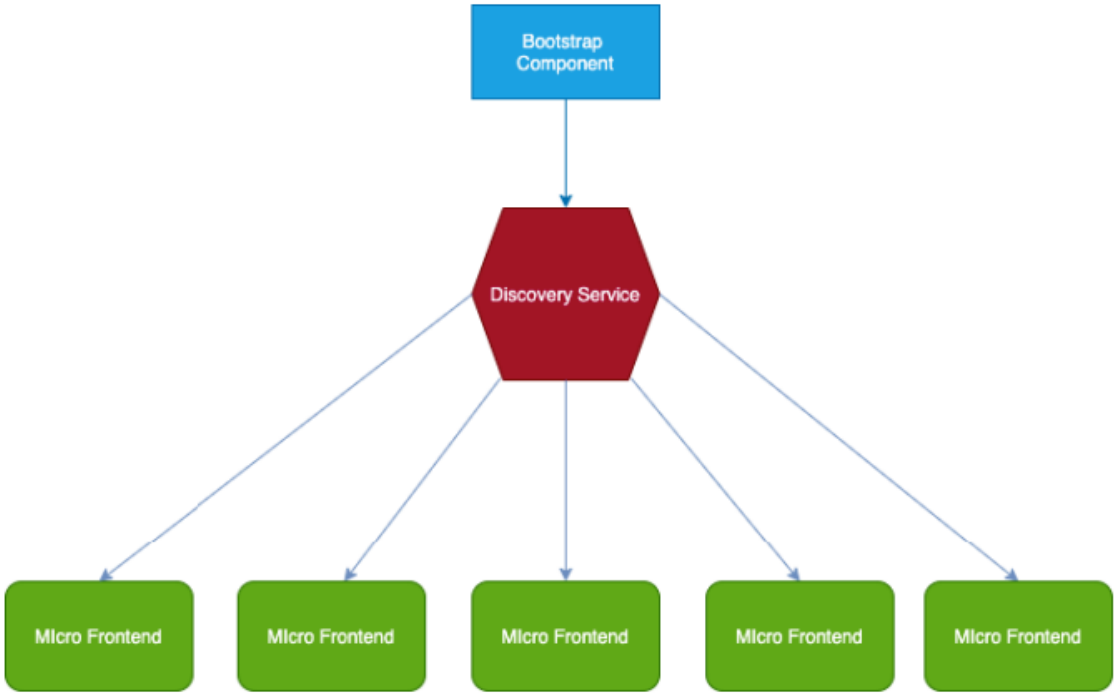

Рисунок 2.3 – Загальна структура архітектури мікрофронтенду

Принцип роботи програми з мікрофронтальною архітектурою полягає в наступному. При запуску програми спочатку запускається компонент Bootsrap, його завдання – визначити, який сервіс потрібно завантажити, потім зробити запит до Discovery Service і отримати дані, необхідні для запуску процесу запуску потрібної служби. Discovery Service є допоміжною програмою, її роль полягає в реєстрації кожної із служб та зберіганні інформації, необхідної для завантаження кожної частини програми.

2.1.4 Вимоги до надійності інформаційної системи

Програмні засоби інформаційної системи повинні забезпечувати:

– Контроль коректності даних, що вводяться, а також контроль несуперечності вхідних даних
– Оповіщення користувача про помилки вхідних даних і суперечливості даних
– Сервери інформаційної системи також повинні мати механізми резервного копіювання та відновлення даних як після збоїв програмного забезпечення, так і в разі збою апаратного забезпечення або відключення електроенергії.

2.1.5 Вимоги щодо безпеки

Система повинна забезпечувати:

– запобігання несанкціонованому доступу до інформації та (або) передачі її особам, які не мають права на доступ до інформації
– застосування механізмів виявлення спроб вторгнення на сайт і отримання несанкціонованого доступу
– розпізнавання типів всіх відомих атак по їх сигнатурам та зберігання їх в окремій базі на сервері
– визначення ступеня важливості сигнатури атаки і налаштування сповіщень або блокування активності в залежності від даного показника
– оповіщення адміністратора сайту про спроби вторгнення на сайт
– надання можливості для адміністратора сайту заблокувати несанкціоновану призначену для користувача активність на сайті.

2.2 Вимоги до розгортання

2.2.1 Постійна розгортка та постійна інтеграція

З боку замовника до системи було висунуто ряд вимог до системи постійної інтеграції та постійної розгортки. Вони мають включати наступні кроки:

- Отримання початкового коду з репозиторію
- Збірка проекту
- Виконання тестів
- Розгортання готового проекту
- Відправлення звітів

Також необхідно розглянути можливість інтеграції із сервісом github, який на даний момент використовується замовником, і репозиторієм системи контролю версій. Кожна розподілена служба матиме власний окремий репозиторій, і всі необхідні кроки повинні бути налаштовані в кожному з них.

Також на етапі постійної інтеграції, щоб мінімізувати витрати ресурсів, необхідно використовувати сторонній сервіс постійної інтеграції. На початковому етапі кроки інтеграції повинні бути безкоштовними для клієнта і вписуватися в запропонований безкоштовним тарифом від постачальника послуг. Була запропонована низка послуг, і після порівняння переваг і недоліків усіх продуктів було обрано CircleCI. Його перевага в тому, що з усіх аналогів він має найшвидший TTM (час виходу на ринок). На рисунку 2.1 показана затверджена система взаємодії зі службою безперервної інтеграції.

Постійна розгортка має проводитись на on-premise сервери які замовник винаймає в компанії Amazon. Проте лише кожну п’ятницю щотижня має буду налаштована інтеграція в production середу. Всі інші розгортки комп’ютерної

системи мають проводитись на тестове середовище і не повинні будь-яким чином впливати на production середу.

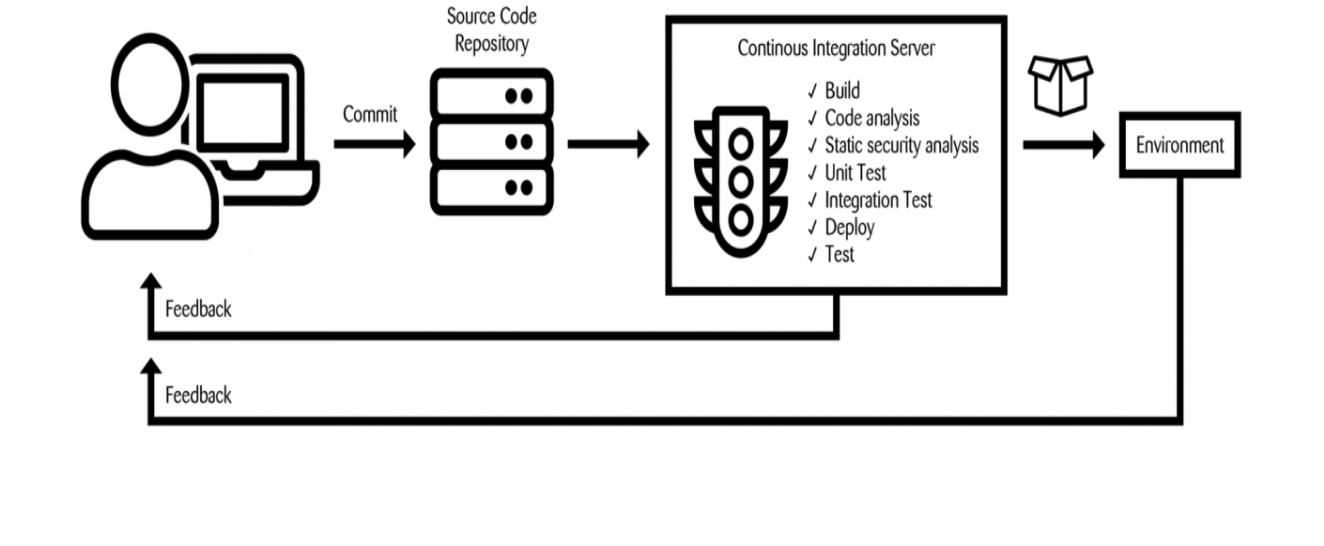

Рисунок 2.4 – Схвалена система взаємодії с сервісом постійної інтеграції

2.2.2 Централізовані обчислювальні платформи

З боку замовника до системи було висунуто вимогу щодо розгортання системи на централізованій обчислювальні платформи. Як платформу для розгортання було обрано сервіс що надається провайдером AWS від компанії Amazon. З використанням цього сервісу необхідно побудувати мікросервісну архітектуру, та мікрофронтенд архітектру на стороні користувацького інтерфейсу та вбудувати готове рішення в існуючу архітектуру з найменшим використанням ресурсів. Для цього буде використано базовий продукт EC2 або EKS. На рисунку 2.5 наведено схему переваг та можливостей сервісу AWS EKS з офіційного сайту.

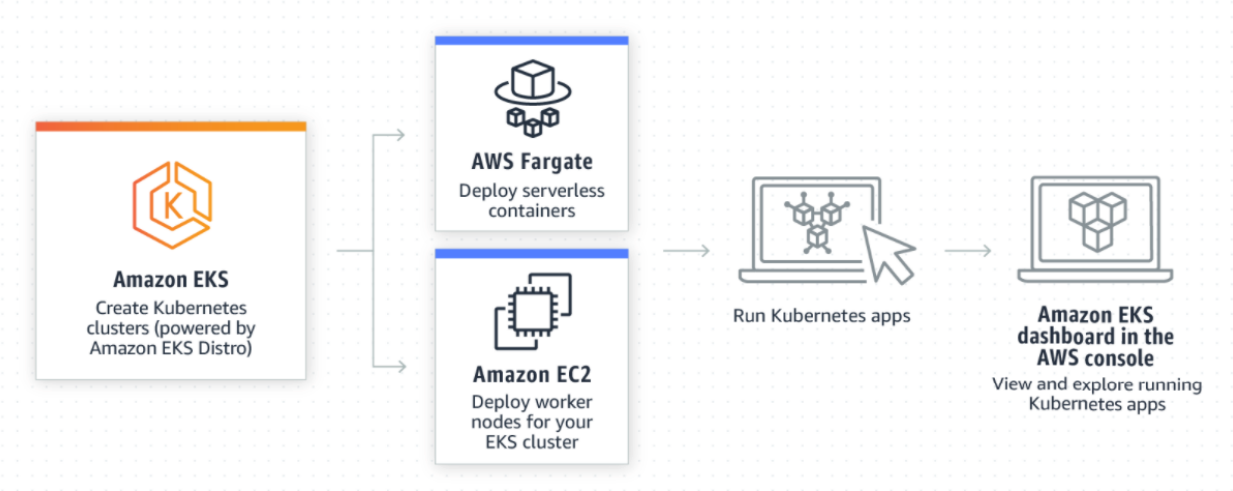

Рисунок 2.5 – Переваги та можливості AWS EKS

На його основі буде працювати оркестратор контейнерів Kebernetes який буде керувати образами контейнерів зібраними за допомогою Docker, які в свою чергу спілкуючись через протокол HTTP вже складають єдину комп’ютерну систему.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Побудова мікросервісів

Архітектура брокера – це один із способів, за допомогою яких ваші послуги можуть спілкуватися між собою. У ньому всі служби оточують сервер обміну повідомленнями, брокер, і всі підключені до нього. Служби надсилають повідомлення брокеру, який потім знає, яка інша служба або послуги йому потрібні для пересилання цих повідомлень. Таким чином, службам не потрібно зберігати інформацію про інші послуги.

Замість цього вони покладаються на брокера, який подбає про всі повідомлення, і це дозволяє їм бути ізольованими та зосередженими лише на своєму конкретному домені. Брокер також може зберігати повідомлення, коли їхні одержувачі не працюють, дозволяючи відправникам і одержувачам не бути примусово підключеними одночасно, що забезпечує ще більшу ізоляцію. Звичайно, у цього рішення є недоліки, оскільки брокер може швидко стати вузьким місцем, оскільки всі комунікації повинні проходити через нього, а також він може стати єдиною точкою збою для вашого бекенда. Однак є кілька способів пом’якшити ці проблеми. Один із способів — запустити паралельно кілька екземплярів брокера, що дозволить кращу стійкість системи до відмов. Іншим способом було б використання інших архітектур. Альтернативні архітектури відрізняються від архітектури, яку ми будемо реалізовувати в цьому посібнику, не використовуючи брокера, або використовуючи іншу архітектуру брокера, або використовуючи інший протокол обміну повідомленнями, такий як HTTP.

Мікросервісний підхід має декілька моделей дизайну архітектури, їх можна розділити на п’ять моделей (рис. 3.1).

Рисунок 3.1 – Патерни мікросервісної архітектури

3.2 Шаблон міграції на мікросервіси

Початок шляху мікросервісів вимагає мінімального рівня операційної готовності. Для цього потрібен доступ до середовища розгортання, створення нових видів безперервних конвеєрів доставки для незалежного створення, тестування та розгортання виконуваних служб, а також здатність захищати, налагоджувати та контролювати розподілену архітектуру. Зрілість оперативної готовності необхідна незалежно від того, створюємо ми нові сервіси чи

розкладаємо існуючу систему. Це включає створення Service Mesh, виділеного інфраструктурного рівня для швидкої, надійної та безпечної мережі мікросервісів, систем оркестровки контейнерів для забезпечення вищого рівня абстракції інфраструктури розгортання та еволюції систем безперервної доставки, таких як GoCD, для створення, тестування та розгортання. мікросервіси як контейнери.

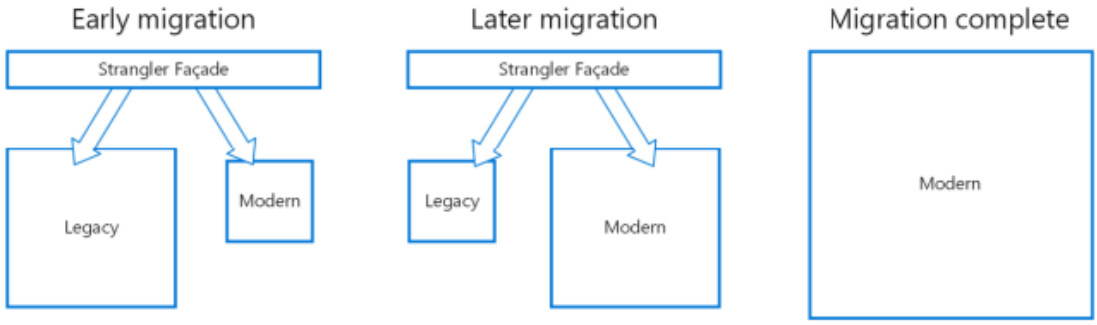

Рисунок 3.2 – Перехід на мікросервісний компонент

В якості основоположного принципу команди доставки повинні мінімізувати залежність новостворених мікросервісів від моноліту. Головною перевагою мікросервісів є швидкий і незалежний цикл випуску. Наявність залежності від моноліту — даних, логіки, API — поєднує службу з циклом випуску моноліту, забороняючи цю перевагу. Часто головною мотивацією відходу від моноліту є висока вартість і повільний темп зміни закріплених у ньому можливостей, тому ми хочемо поступово рухатися в напрямку, який роз’єднує ці основні можливості, видаляючи залежності від моноліту. Якщо команди дотримуються цієї інструкції, розбудовуючи можливості у своїх власних сервісах, замість цього вони знаходять залежності у зворотному

напрямку, від моноліту до сервісів. Це бажаний напрямок залежності, оскільки він не уповільнює темпи змін для нових послуг.

3.3 Робота з образами

3.3.1 Створення образів

Образ Docker — це інертний шаблон, доступний лише для читання, який містить інструкції щодо розгортання контейнерів. У Docker все в основному обертається навколо зображень.

Зображення складається з набору файлів (або шарів), які об’єднують усе необхідне — такі як залежності, вихідний код та бібліотеки — необхідні для створення повністю функціонального середовища контейнера.

Зображення зберігаються в реєстрі Docker, як-от Docker Hub, або в локальному реєстрі. Образ Docker складається з кількох шарів.

Шари зображення накладаються один на одного. Хоча шари можуть відрізнятися один від одного, кожен із них може залежати від шару, який знаходиться безпосередньо під ним.

Ці шари, також звані проміжними зображеннями, по суті є файлами лише для читання, до яких після налаштування віртуалізованого середовища буде додано шар-контейнер.

Ви можете використовувати команду docker images, щоб переглянути деталі зображення у вашій системі.

Крім того, ви також можете використовувати команду docker history, щоб побачити всі шари, які складають завантажене зображення.

Кожен шар є інструкцією з Dockerfile. Розглянемо наступний файл (рис. 3.3).

```
FROM ubuntu:15.04

COPY . /app

RUN make /app

CMD python /app/app.py
```

Рисунок 3.3 – Приклад Dockerfile

Цей Dockerfile має чотири команди, кожна команда зі списку створює шар. Образи та контейнери Docker працюють разом, щоб дозволити вам розкрити весь потенціал інноваційної технології Docker. Однак вони мають незначні відмінності, які може бути важко помітити, особливо новачкові.

Проста аналогія, яка порівнює їх відмінності, полягає в тому, щоб уявити зображення Docker як рецепт, а контейнер — як торт, приготований за цим рецептом.

Рецепт містить інструкцію з випікання торта. Ви не можете насолоджуватися їсти торт, якщо не виконуєте інструкції.

Щоб приготувати торт і з’їсти його, потрібно дотримуватися рецепта. Аналогічно, ви повинні дотримуватись інструкцій у образі Docker, щоб створити та запустити контейнер і насолоджуватися перевагами Docker.

Ви можете спекти якомога більше тортів за одним рецептом — так само, як зображення може створити кілька контейнерів. Однак, якщо ви змініте рецепт, смак ваших існуючих тортів не зміниться.

За зміненим рецептом використовуватимуться тільки щойно випечені торти. Аналогічно, якщо ви внесете зміни до зображення контейнера, ви не вплинете на вже запущені контейнери. Зображення Docker — це шаблони лише для читання, які використовуються для створення контейнерів. Контейнери — це розгорнуті екземпляри, створені на основі цих шаблонів. Зображення та

контейнери тісно пов’язані між собою і є важливими для забезпечення живлення програмної платформи Docker.

3.3.2 Реєстр образів

Реєстр Docker — це система зберігання та розповсюдження іменованих образів Docker. Одне і те саме зображення може мати кілька різних версій, ідентифікованих за допомогою тегів.

Реєстр Docker організований у репозиторії Docker, де репозиторій містить усі версії певного образу. Реєстр дозволяє користувачам Docker витягувати зображення локально, а також надсилати нові зображення до реєстру (за наявності відповідних дозволів на доступ, якщо це можливо).

За замовчуванням механізм Docker взаємодіє з DockerHub, екземпляром публічного реєстру Docker. Однак можна запустити локально відкритий реєстр/дистрибутив Docker, а також комерційно підтримувану версію під назвою Docker Trusted Registry. Є й інші публічні реєстри, доступні в Інтернеті.

DockerHub — це розміщене рішення для реєстру від Docker Inc. Окрім публічних та приватних сховищ, воно також забезпечує автоматичне збирання, облікові записи організацій та інтеграцію з рішеннями для контролю джерел, такими як Github та Bitbucket. DockerHub може виконувати автоматичне збирання образів, якщо репозиторій DockerHub пов’язано зі сховищем керування кодом, яке містить контекст збірки (Dockerfile та всі файли в одній папці). Коміт у вихідному репозиторії ініціює збірку в DockerHub. Зауважте, що кілька приватних репозиторій, паралельні збірки та сканування безпеки зображень доступні лише за платних підписок. DockerHub також може автоматично сканувати зображення в приватних сховищах на наявність вразливостей, створюючи звіт із детальним описом уразливостей, знайдених у кожному шарі зображень, за серйозністю (критичні, великі чи незначні).

3.4 Теорія роботи контейнерів

3.4.1 Запуск контейнерів

Подумайте про контейнер як про іншу форму віртуалізації. Віртуальні машини, також лише одна з форм віртуалізації, дозволяють апаратній частині розміщувати декілька операційних систем як програмне забезпечення. Віртуальні машини додаються до хост-машини, щоб апаратна потужність могла розподілятися між різними користувачами і відображатися як окремі сервери або машини. Контейнери віртуалізують ОС, розбиваючи її на віртуалізовані відсіки для запуску контейнерних додатків.

І контейнер із відкритим кодом Docker, і підхід компанії привабливі, особливо для хмарних додатків та загального розвитку

Демон Docker — це те, за що насправді відповідає збирання та виконання коду, а також розповсюдження завершених контейнерів. Він приймає команди, які розробник вводить у клієнтський термінал Docker, і виконує їх.

Такий підхід дозволяє поміщати фрагменти коду в менші, легко транспортувальні частини, які можна запускати в будь-якому місці, де працює Linux або Windows. Це спосіб зробити програми ще більш розподіленими та розділити їх на конкретні функції.

Розглянемо команду для запуску контейнеру (рис 3.4).

```
docker run -i -t ubuntu /bin/bash
```

Рисунок 3.4 – Приклад команди для запуску

На цьому закінчується швидкий огляд потужної команди docker run, яка, швидше за все, буде командою, яку ви будете використовувати найчастіше. Має сенс витратити деякий час, щоб почуватись комфортно. Щоб дізнатися

більше про run, скористайтеся docker run --help, щоб побачити список усіх прапорів, які він підтримує. Продовжуючи далі, ми побачимо ще кілька варіантів запуску docker.

Але перш ніж рухатися далі, давайте швидко поговоримо про видалення контейнерів. Вище ми бачили, що ми все ще можемо бачити залишки контейнера навіть після виходу, запустивши docker ps -a. Протягом цього підручника ви будете запускати docker run кілька разів, а залишення випадкових контейнерів буде споживати місце на диску. Тому, як правило, я прибираю контейнери, коли закінчую з ними. Для цього ви можете запустити команду docker rm.

В останньому розділі ми використовували багато жаргону, характерного для Docker, який може ввести деяких з пантелику. Тому, перш ніж йти далі, дозвольте мені пояснити деяку термінологію, яка часто використовується в екосистемі Docker.

Контейнери - створюються з образів Docker і запускають саму програму. Ми створюємо контейнер за допомогою докера, який ми зробили за допомогою завантаженого образу busybox. Список запущених контейнерів можна побачити за допомогою команди docker ps.

Docker Daemon – фонова служба, що працює на хості, яка керує створенням, запуском та розповсюдженням контейнерів Docker. Демон — це процес, який виконується в операційній системі, з якою спілкуються клієнти.

Docker Client – інструмент командного рядка, який дозволяє користувачеві взаємодіяти з демоном. Загалом, можуть бути й інші форми клієнтів, наприклад Kitematic, які надають користувачам графічний інтерфейс.

Docker Hub — реєстр образів Docker. Ви можете думати про реєстр як про каталог усіх доступних образів Docker. Якщо потрібно, можна розмістити власні реєстри Docker і використовувати їх для отримання зображень.

3.4.2 Кластери контейнерів

Режим Docker Swarm дає змогу керувати кластером Docker Engines, створеним на платформі Docker. Ви можете використовувати CLI Docker для створення зграї, розгортання служб додатків у рою та керування поведінкою роя.

Swarm може допомогти розробникам та IT-адміністраторам. Координуйте між контейнерами та розподіляйте завдання групам контейнерів. Виконуйте перевірку працездатності та керуйте життєвим циклом окремих контейнерів. Забезпечте резервування та відмову в разі відмови вузлів. Збільшуйте і зменшуйте кількість контейнерів залежно від навантаження. Виконуйте поточні оновлення програмного забезпечення в кількох контейнерах

3.5 Збереження даних

За замовчуванням усі файли, створені всередині контейнера, зберігаються на шарі контейнера для запису. Це означає що:

Дані не зберігаються, коли цього контейнера більше не існує, і може бути важко отримати дані з контейнера, якщо це потребує інший процес.

Шар для запису контейнера тісно пов’язаний з хост-машиною, на якій працює контейнер. Ви не можете легко перемістити дані в інше місце.

Для запису в шар для запису контейнера потрібен драйвер сховища для керування файловою системою. Драйвер сховища забезпечує файлову систему об’єднання, використовуючи ядро Linux. Ця додаткова абстракція знижує продуктивність у порівнянні з використанням томів даних, які записують безпосередньо в файлову систему хоста.

Docker має два варіанти зберігання файлів у контейнерах на хост-машині, щоб файли зберігалися навіть після зупинки контейнера: томи та монтування прив’язок. Якщо ви використовуєте Docker в Linux, ви також можете використовувати монтування tmpfs. Якщо ви використовуєте Docker у Windows, ви також можете використовувати іменований канал.

3.7 Розгортка застосунку в мережі

Групи безпеки дозволяють вам контролювати трафік до вашого екземпляра, включаючи тип трафіку, який може досягати вашого екземпляра. Наприклад, ви можете дозволити комп’ютерам лише з вашої домашньої мережі отримувати доступ до вашого екземпляра за допомогою SSH. Якщо ваш екземпляр є веб-сервером, ви можете дозволити всім IP-адресам отримати доступ до вашого екземпляра за допомогою HTTP або HTTPS, щоб зовнішні користувачі могли переглядати вміст вашого веб-сервера.

Вирішіть, кому потрібен доступ до вашого екземпляра; наприклад, один хост або конкретна мережа, якій ви довіряєте, наприклад публічна IPv4-адреса вашого локального комп’ютера. Редактор групи безпеки на консолі Amazon EC2 може автоматично визначити публічну IPv4-адресу вашого локального комп’ютера.

Групи безпеки за замовчуванням і щойно створені групи безпеки містять правила за замовчуванням, які не дають вам отримати доступ до вашого екземпляра з Інтернету. Щоб увімкнути доступ до мережі до вашого екземпляра, ви повинні дозволити вхідний трафік до вашого екземпляра. Щоб відкрити порт для вхідного трафіку, додайте правило до групи безпеки, яку ви пов’язали зі своїм екземпляром під час його запуску.

Щоб підключитися до вашого екземпляра, ви повинні налаштувати правило для авторизації трафіку SSH з загальнодоступної адреси IPv4 вашого

комп’ютера. Щоб дозволити трафік SSH з додаткових діапазонів IP-адрес, додайте ще одне правило для кожного діапазону, який потрібно авторизувати.

Якщо ви ввімкнули свій VPC для IPv6 і запустили свій екземпляр з адресою IPv6, ви можете підключитися до свого екземпляра, використовуючи його адресу IPv6 замість загальнодоступної адреси IPv4. Ваш локальний комп’ютер повинен мати адресу IPv6 і повинен бути налаштований на використання IPv6.

Ось таблиця з порівнянням послуг, які пропонують постачальники віртуального хостингу та хмарних провайдерів для розгортання додатків:

Таблиця 3.1 – Порівняння характеристик хостингу та хмарних сервісів

	Хостинг	Хмара
Ресурси пам’яті та Процесора	Поділенні між користувачами	Ізольовані між собою
Вибір ОС	Тільки з запропонованих	Безліч, навіть власну
Надійність	Неможливо визначити	Висока
Віртуалізація	Віртуальні машини	Контейнери на різніх серварах
Масштабованість	Пропонується хостом Складно реалізувати	Автоматична за планувальником
Вартість	Низька	Середня
Виділення ресурсів	Довгий процес	Швидко

Окрім економічності, використання мікросервісів надає такі переваги.

Коротший час виходу на ринок. Коли різні команди можуть розробляти свої мікросервіси незалежно, у всьому продукті буде менше перешкод, а загальний час виходу на ринок зменшується.

Покращена масштабованість. Коли попит на певні функції зростає або зменшується, ви можете збільшувати і зменшувати їх окремо для кількох серверів і постачальників.

Стійкість операцій. Оскільки мікросервіси працюють незалежно один від одного, збій одного компонента не вимкне весь продукт, як це відбувається з монолітними додатками, а несправний модуль легко перезавантажити.

Ефективність управління. Модульні програмні продукти на основі мікросервісів набагато легше розгортати, оновлювати та керувати, це безумовно їх основна перевага.

Простота оцінки товару. Коли монолітні програми розбиті на незалежні частини, зрозуміти вихідний код кожної мікросервіси набагато легше, тому розробники можуть оцінювати та оновлювати їх набагато швидше.

3.7 Забезпечення мережевого доступу до додатку

Службі мають файли маніфестів, що описують їх поведінку та що від них очікує користувач.

За допомогою Cloud Shell створимо маніфест для Kubernetes сервісу, назвемо його, зазначаємо команду apiVersion та вказуємо параметр v1, встановлений тип серсвіс, и вказуємо назву контейнеру.

Необхідно в маніфесті вказати тип сервісу (рис. 3.5).

```
spec:
  type: ClusterIP
```

Рисунок 3.5 – Зазначення типу сервісу у маніфесті

Також необхідно визначити елементи, що будуть групуватися службою, це віконується наступними рядками(рис. 3.6).

```
  selector:
    app: example-website
```

Рисунок 3.6 – Зазначення под у маніфесті

Далі необхідно для мережевих налаштувань назначити порт та протокол для маршрутизації запитів (рис 3.7).

```
ports:
    - port: 80
      name: http
     targetPort: http
     protocol: TCP
```

Рисунок 3.7 – Зазначення портів та протоколів у маніфесті

Далі після оновлення конфігураційного файлу необхідно зазначити що ми будемо його використовувати, зробимо це:

kubectl apply -f ./service.yaml

Далі, отримаємо наступну конфігурацію кластера:

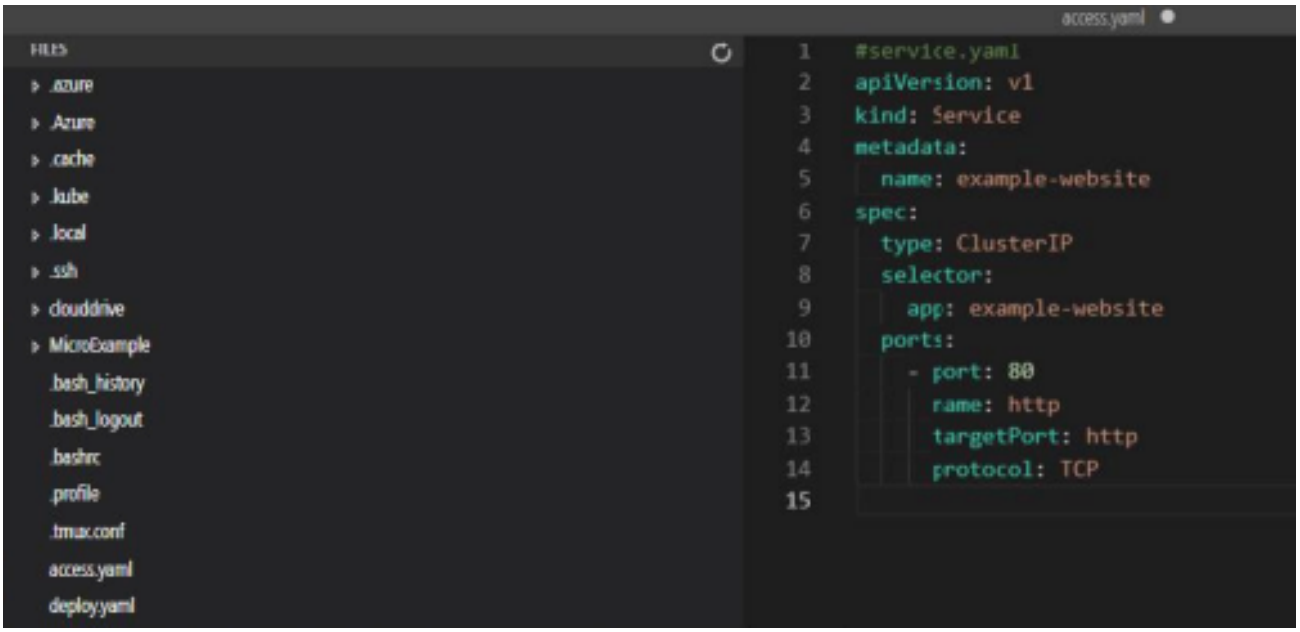

Рисунок 3.8 – Конфігурація кластера

Далі налаштуємо додатку постійний доступ в інтернет за допомогою сервісу DNS. Створимо вхідний контролер, та файл маніфесту, що матиме право на вихід в мережу (рис. 3.9).

```
#ingressExample.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: Example-website
```

Рисунок 3.9 – Конфігурація контролеру входу

Зазначимо додаткові атрибути маршрутизації потоків даних через HTTP, за це відповідає атрибут annotations, в метаданих встановимо необхідне

значення для поля kubernetes.io/ingress.class, наразі це значення addon-http-application-routing

```
annotations:
    kubernetes.io/ingress.class: addon-http-application-routing
```

Рисунок 3.10 – Налаштування атрибуту атрибута annotations

Наступним кроком – необхідно визначити і встановити параметри хоста під тегом специфікації, і щоб хост був унікальним і повязаним з ресурсною групою необхідно згенерувати унікальну назву зони для конкретного кластера за допомогою команди:

Далі треба конфігурувати хост за допомогою тегу специфікації, також для нього необхідно згенерувати назву зону та локалі для обраного кластера, щоб він був унікальним в своєму регіоні це виконується за допомогою наступної команди з опціями:

az network dns zone list --output table

Далі конфігуруємо правила входу для клієнтів та шляхи за якими буде надаватися доступ. Для параметру http додамо обов’язкові системні данні paths (рис 3.11).

```
      http:
        paths:
          - backend: # rules to handle requets
              serviceName: Example-website
              servicePort: http
            path: /
```

Рисунок 3.11 – Налаштування paths

Коли розгортка успішно скінчена, необхідно переконатися що не виникло збоїв:

kubectl get ingress stuff-manager

результат команди матиме повну IP адресу за яким налаштовано досту до кластера, а також системну інформацію – налаштування DNS, розташування кластеру в мережевій архітектурі, його таблицю зовнішніх доступів, та список додаткових, резервних хостів які було автоматично створено задля забезпечення стабільності. Тепер можем з боку кінцевого користувача перевірити застосунок.

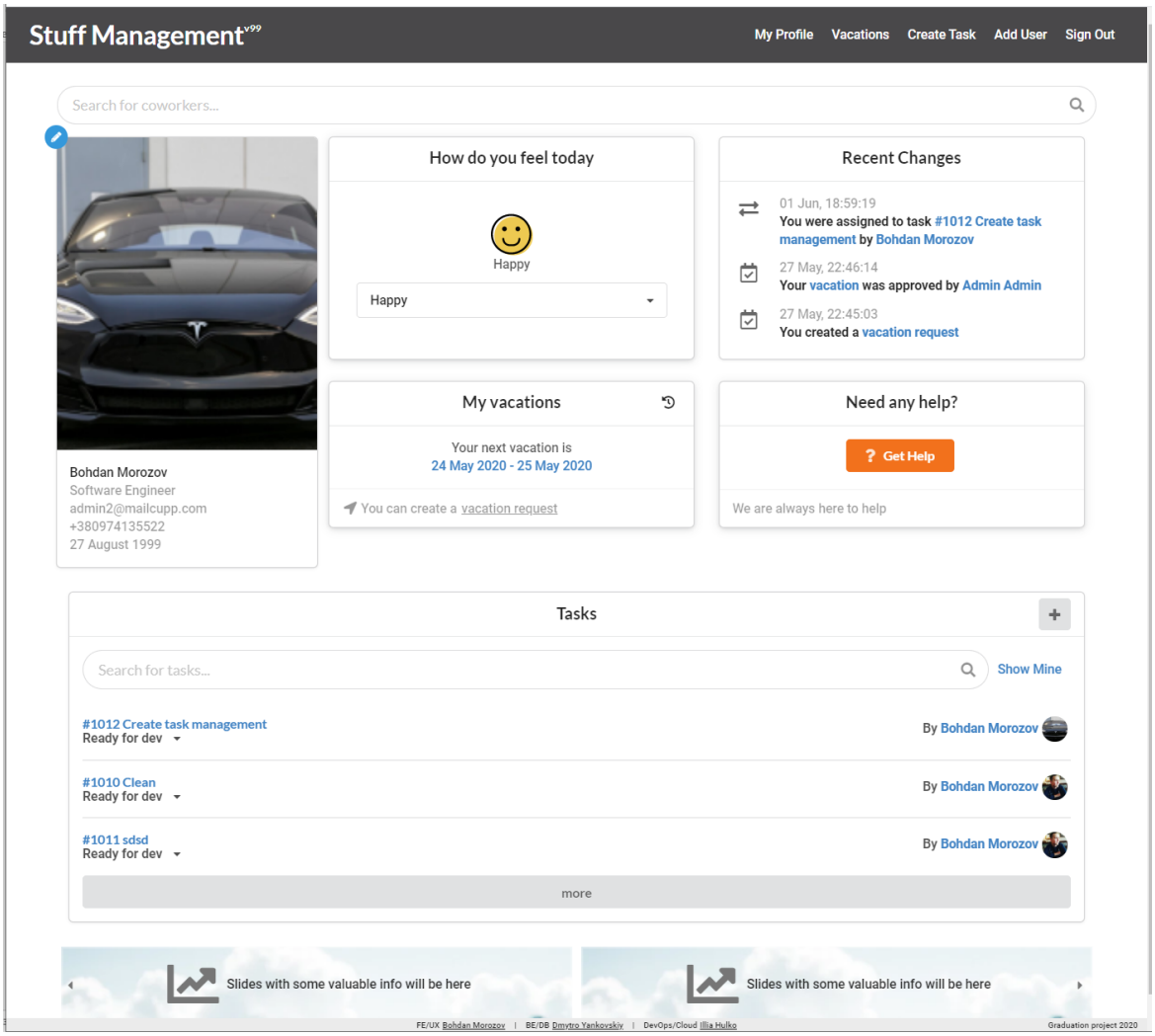

Рисунок 3.12 – Застосунок доступний в мережі

3.8 Створення кластера Kubernetes

Amazon Web Services надає три основні варіанти розгортання Kubernetes:

- Запуск Kubernetes безпосередньо на машинах Amazon EC2
- Використання служби Amazon Elastic Kubernetes (EKS)
- Використання kops—системи забезпечення з відкритим кодом, створеної для AWS, яка надається як частина проекту Kubernetes.

Ми зосередимося на перших двох варіантах. Щоб дізнатися більше про kops, перегляньте офіційну документацію.

Служба Amazon Elastic Kubernetes (Amazon EKS) дає змогу розгортати Kubernetes і керувати ним на AWS без необхідності запускати Kubernetes безпосередньо на машинах EC2, як ми показали вище. EKS сертифікований проектом Kubernetes, тому наявні програми, інструменти та плагіни з екосистеми Kubernetes повинні працювати правильно.

Amazon Elastic Container Service for Kubernetes (EKS) — це повністю керована служба, яка займається налаштуванням і створенням кластера, забезпечуючи підтримку кількох AZ для всіх кластерів і автоматичну заміну непрацездатних екземплярів (головних чи робочих вузлів). Він також виправляє та оновлює кластери до останнього рекомендованого випуску Kubernetes, не вимагаючи будь-якого втручання.

Хоча EKS забезпечує подібний рівень інтеграції з іншими сервісами Amazon, як і ECS, він спирається на відкриту модель оркестрування Kubernetes, а не на специфічну модель AWS. Це збільшує переносимість кластерів, розгорнутих на EKS, до інших хмарних постачальників. Основним аргументом для такої міграції буде рівень з’єднання з нативними службами AWS, але принаймні сторона оркестрування буде легкою.

Робочі вузли запускаються на власних інстанціях EC2 користувача AWS, тому не надаються іншим клієнтам. Щоб використовувати такі інструменти, як kubectl, доступ до головних екземплярів має бути налаштований через

загальнодоступні кінцеві точки IAM або через AWS PrivateLink. З AWS PrivateLink головні пристрої виглядають як еластичний мережевий інтерфейс із приватними IP-адресами в Amazon VPC. Це дозволяє майстрам і службі EKS безпосередньо з Amazon VPC, не використовуючи публічні IP-адреси та не вимагаючи трафіку для проходження Інтернету.

Amazon EKS також тісно інтегрується з іншими службами AWS, такими як ELB для балансування навантаження або AWS CloudTrail для ведення журналів.

Коли кластер буде готовий, контейнерні програми можна розгорнути за допомогою каталогу програм Rancher або kubectl. kubectl потрібно налаштувати через інтерфейс користувача Rancher, щоб інформація про розгортання стала видимою на інформаційних панелях Rancher.

Terraform — це інструмент інфраструктури як коду, який використовується для безпечного та ефективного створення, зміни та керування версіями інфраструктури. Його можна використовувати для розгортання контейнерних додатків у належним чином налаштований кластер Kubernetes, що працює в AWS. Terraform використовує власну мову конфігурації і за замовчуванням шукає специфікації ресурсів у тому самому каталозі, де виконуються команди terraform. Також AWS надає можливість налаштовувати кластери в EKS через інтерфейс користувача, але це не є оптимальним с точки зору автоматизації.

Ви можете використовувати наступні команди, щоб встановити ім’я хоста на кожному сервері. Після виконання наступних команд на кожному сервері повторно увійдіть на сервери, щоб сервери отримали нове ім’я хосту.

sudo hostnamectl set-hostname "master"

sudo hostnamectl set-hostname "node1"

sudo hostnamectl set-hostname "node2"

Далі необхідно отримати ключ, додати репозиторій Kubernetes, та приєднати робочі вузли до кластера. Кожен вхідний контролер включає в себе проксі-сервер, який є зворотнім. Він автоматично обробляє всі запити на DNS через єдину точку входу. Не треба щоразу налаштовувати DNS запис разом з розгорткою нових служб, це автоматизована робота контролера. Кожного разу коли на кластері реєструється новий вхід, контролер створює новий DNS запис у зоні під управлінням AWS і зразу пов’язує його с необхіднім балансувальним сервером. Все це забезпечує максимальне спрощення процесу надання мережевого доступу.

Для початку створимо групу кластерів за використовуючи команду az group create, вказавши обовязкові параметри:

az group create --named rg-sample mate --location eastus

Надалі в цій групі будемо створювати вже безпосередньо самі кластери, вказавши їх параметри (рис 3.13).

```
az aks create \
    --resource-group rg-MicroExample \
    --name aks-MicroExample \
    --node-count 2 \
    --enable-addons http_application_routing \
    --dns-name-prefix example-kubernetes-$RANDOM \
    --generate-ssh-keys \
        --node-vm-size Standard_B2s
```

Рисунок 3.13 – Команда створення кластеру

За допомогою команди az aks add створимо кластер з назвою aks-sample в групі яку ми налаштували в попередніх кроках, в ній буде декілька нод, до прикладу дві, маршрутизацію реалізуємо з http_appl_rout до DNS записів, а розмір віртуальної машини залишимо Standard.

Наступне – необхідно налаштувати зворотній зв'язок кластера та його групи (рис 3.14).

```
az aks get-credentials --name aks-MicroExample --resource-group rg-MicroExample
```

Рисунок 3.14 – Налаштування групи кластера

Об’єкти Kubernetes є постійними сутностями в системі Kubernetes. Kubernetes використовує ці сутності для представлення стану вашого кластера. У файлі .yaml для об’єкта Kubernetes, який ви хочете створити, вам потрібно встановити значення для таких полів:

– apiVersion – яку версію API Kubernetes ви використовуєте для створення цього об’єкта
– kind - який об'єкт ви хочете створити метадані - дані, які допомагають однозначно ідентифікувати об'єкт, включаючи рядок імені, UID і необов'язковий простір імен
– spec - який стан ви бажаєте для об'єкта
– desc – зрозуміли для користувача короткий опис об’єкта який налаштовується

Зокрема, файл маніфесту повинен містити основні елементи зображені на рисунку 3.15.

```
apiVersion: apps/v1
kind: Deployment
metadata:
        name: micro-website # the name of the deployment
```

Рисунок 3.15 – Основні параметри Kubernetes маніфесту

Визначаючи конфігурації, вкажіть останню стабільну версію API.

Конфігураційні файли слід зберігати в системі контролю версій, перш ніж передавати їх у кластер. Це дозволяє швидко відкотити зміну конфігурації, якщо це необхідно. Це також сприяє відтворенню та відновленню кластера.

Напишіть файли конфігурації, використовуючи YAML, а не JSON. Хоча ці формати можна використовувати як взаємозамінні майже в усіх сценаріях, YAML, як правило, більш зручний для користувачів.

Зауважте також, що багато команд kubectl можна викликати в каталозі. Наприклад, ви можете викликати kubectl apply до каталогу файлів конфігурації:

kubectl apply -f ./deploy.yaml

Не вказуйте значення за замовчуванням без потреби: проста мінімальна конфігурація зменшить ймовірність помилок. Помістіть описи об’єктів в анотації, щоб забезпечити кращий самоаналіз. Далі створимо образ за допомогою az acr create (рис. 3.16). Та збережемо його в AWS Container Registry.

```
az acr create \
--resource-group rg-MicroExample \
--name ACRMicroExample\
    --sku Basic
```

Рисунок 3.16 – Створення образу контейнера

Так після переходу в директорію програми виконаємо команду az acr build (рис. 3.17).

```
bohdan_morozov@EPUADNIW010E customer-app % cd ..
bohdan_morozov@EPUADNIW010E customer-app % az acr build --image customer-app --registry
```

Рисунок 3.17 – Виконання команд розгортки

Мітки — це пари ключ/значення, які прикріплені до об’єктів, наприклад модулів. Мітки призначені для використання для визначення ідентифікаційних атрибутів об’єктів, які є значущими та релевантними для користувачів, але не мають прямого на увазі семантику для основної системи. Мітки можна використовувати для організації та вибору підмножин об’єктів.

Мітки можна прикріплювати до об’єктів під час створення, а потім додавати та змінювати в будь-який час. Кожен об’єкт може мати певний набір міток ключ/значення. Кожен ключ повинен бути унікальним для даного об’єкта. Розглянемо створення мітки (рис. 3.18).

```
spec:
template: # Шаблон пода деплоймента
  metadata: # Метадані pod
  labels:
        app: Example-website
```

Рисунок 3.18 – Вказання тегу label

У файлі конфігурації ви можете побачити, що Pod має том, який спільно використовують контейнер ініціалізації та контейнер програми (рис 3.18). Контейнер ініціації монтує спільний том у /work-dir, а контейнер програми монтує спільний том у /usr/share/nginx/html.

```
spec:
containers:
- image: <acr_name>.azurecr.io/Example-website
name: contoso-website
```

Рисунок 3.19 – Вказання посилання на розгорнутий сайт

Рекомендується розміщувати ресурси, пов’язані з одним і тим же мікросервісом або рівнем програми, в один файл і групувати всі файли, пов’язані з вашою програмою, в одному каталозі. Якщо рівні вашої програми зв’язуються один з одним за допомогою DNS, ви можете розгорнути всі компоненти свого стека разом.

Встановимо значення лімітів для використання ресурсів (рис 3.19).

```
resources:
      requests:
        cpu: 50m
        memory: 128Mi
      limits:
        cpu: 200m
              memory: 256Mi
```

Рисунок 3.20 – Затвердження ліміту використання ресурсів

А також порта, на якому у поді буде розміщено реусрс:

```
   ports:
         - containerPort: 80
```

Рисунок 3.21 – Затвердження ліміту використання ресурсів

Для збереження налаштованої конфігурації знову використаємо команду:

kubectl apply

Далі перевіряємо чи відбулися помилки під час запуску, а також переглянемо базову інформації по подах, за допомогою:

kubectl get pods

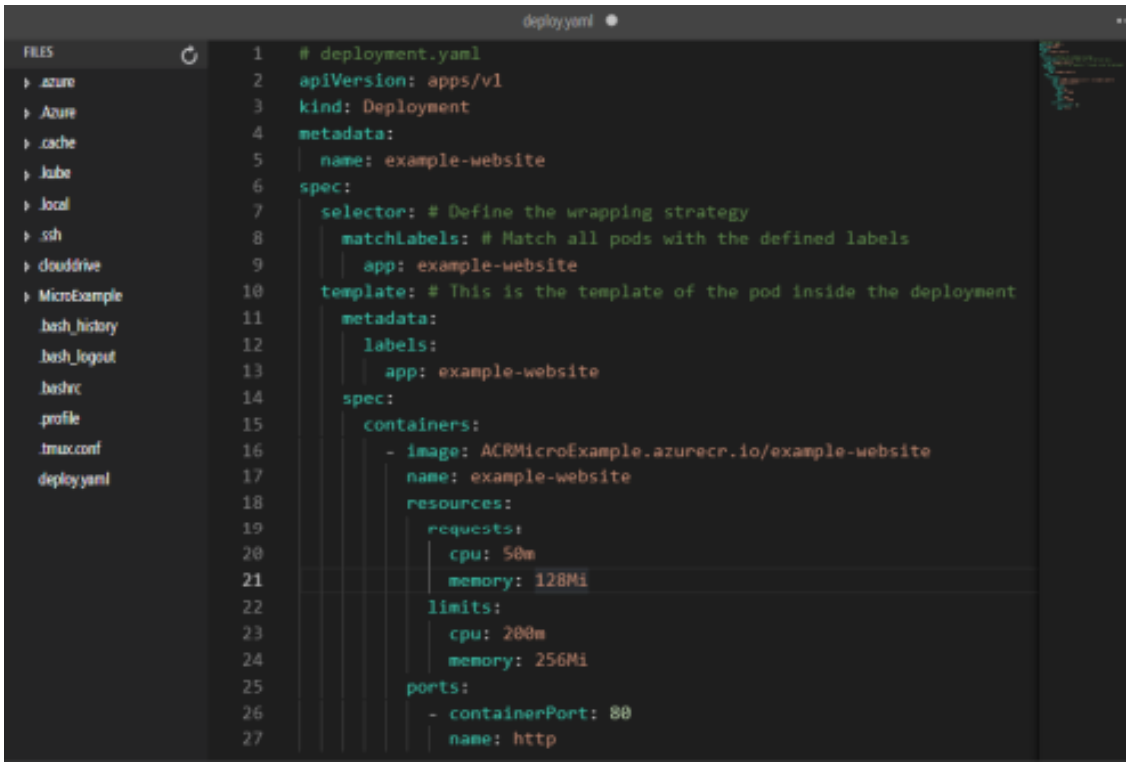

Рисунок 3.22 – Фінальний файл конфігурації

```
NAME                              READY   STATUS             RESTARTS   AGE
example-website-75bfc74bc-cx764   0/1     ImagePullBackOff   0          2m29s
```

Рисунок 3.23 – Результат перевірки

3.9 Налаштування автоматичного мастштабування

Kubernetes підтримує горизонтальне автоматичне масштабування. Горизонтальне масштабування означає, що реакція на збільшення навантаження полягає в розгортанні більшої кількості модулів. Це відрізняється від вертикального масштабування, яке для Kubernetes означало б призначення додаткових ресурсів (наприклад: пам’яті або ЦП) для Pods, які вже запущені для робочого навантаження.

Якщо навантаження зменшується, а кількість модулів перевищує налаштований мінімум, HorizontalPodAutoscaler дає вказівку ресурсу робочого навантаження (Deployment, StatefulSet або інший подібний ресурс) зменшити масштаб. Ці запити та обмеження ресурсів визначаються, як показано у фрагменті коду файлу маніфесту (рис. 3.13).

```
resources:
  requests:
     cpu: 200m /min percentage requested
  limits:
     cpu: 500m /limit to scale
```

Рисунок 3.24 – Визначені обмеження ресурсів

Тепер, коли сервер запущено, створіть автоматичний масштабатор за допомогою kubectl. Існує підкоманда kubectl autoscale, частина kubectl, яка допомагає вам це зробити.

Незабаром ви запустите команду, яка створює HorizontalPodAutoscaler, який підтримує від 1 до 10 копій Pods, контрольованих за допомогою розгортання php-apache, створеного вами на першому кроці цих інструкцій.

Грубо кажучи, контролер HPA буде збільшувати та зменшувати кількість реплік (за допомогою оновлення Deployment), щоб підтримувати середнє використання ЦП для всіх модулів на рівні 50%. Потім розгортання оновлює ReplicaSet — це частина того, як усі розгортання працюють у Kubernetes — а потім ReplicaSet додає або видаляє Pods на основі змін у його .spec.

Оскільки кожен модуль запитує 200 за допомогою kubectl, це означає, що середнє використання ЦП становить 100

kubectl autoscale deployment aks-MicroExample --cpu-percent=75 --min=4 --max=9

Далі подивіться, як автомасштабувальник реагує на збільшення навантаження. Для цього ви запустите інший Pod, який буде виконувати роль клієнта:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: aks-MicroExample
spec:
  maxReplicas: 9
  minReplicas: 4
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: aks-MicroExample
  targetCPUUtilizationPercentage: 75
```

Рисунок 3.25 – Налаштування autoscaling маніфесту

Як і попередні файли маніфесту, цей також необхідно визначити як основний щоб він був вікористани, зробити це можна через інтерфейс користувача в системі AWS EKS, або скористатися універсальною альтернативою на всі часи, та виконати від імені адміністратора на віддаленому кластері наступні команди:

kubectl apply -f scale - hpa.yaml

kubectl apply -f scale – asc-group.yaml

Зверніть увагу, що поле targetCPUUtilizationPercentage було замінено масивом під назвою metrics. Показник використання ЦП є показником ресурсу, оскільки він представлений у відсотках від ресурсу, зазначеного в контейнерах pod. Зверніть увагу, що ви можете вказати інші показники ресурсу, окрім ЦП. За замовчуванням єдиним підтримуваним показником ресурсу є пам’ять. Ці ресурси не змінюють назви від кластера до кластера і повинні бути завжди доступними, якщо доступний API metrics.k8s.io.

Ви також можете вказати показники ресурсу у вигляді прямих значень, а не у відсотках від запитаного значення, використовуючи target.type AverageValue замість Utilization та встановлюючи відповідне поле target.averageValue замість target.averageUtilization.

Є ще два типи показників, обидва з яких вважаються користувацькими показниками: метрики модулів і показники об’єктів. Ці показники можуть мати назви, які залежать від кластера, і вимагатиме більш розширеного налаштування моніторингу кластера.

Перший з цих альтернативних типів метрик — це метрики pod. Ці показники описують модулі, їх усереднюють разом із пакетами та порівнюють із цільовим значенням, щоб визначити кількість реплік. Вони працюють так само, як показники ресурсів, за винятком того, що підтримують лише цільовий тип AverageValue. Другий альтернативний тип метрики — об’єктні метрики. Ці показники описують інший об’єкт в тому самому просторі імен, замість того, щоб описувати модулі. Показники не обов'язково витягуються з об'єкта; вони це лише описують. Показники об’єктів підтримують цільові типи як Value, так і AverageValue. За допомогою Value ціль порівнюється безпосередньо з показником, отриманим від API. З AverageValue значення, яке повертає API користувацьких показників, ділиться на кількість модулів перед порівнянням із цільовим значенням.

3.10 Розгортання контейнерного додатку

У цій моделі розгортання компоненти Kubernetes працюють у вигляді кількох контейнерів:

- etcd: цей компонент зберігає дані конфігурації, до яких може отримати доступ API-сервер Kubernetes Master за допомогою простого HTTP або JSON API.
- Сервер API: цей компонент є центром керування для головного вузла Kubernetes. Це полегшує зв'язок між різними компонентами, тим самим підтримуючи здоров'я кластера.
- Диспетчер контролера: цей компонент гарантує, що бажаний стан кластерів відповідає поточному стану, масштабуючи робочі навантаження вгору і вниз.
- Планувальник: цей компонент розміщує робоче навантаження на відповідний вузол – у цьому випадку всі робочі навантаження будуть розміщені локально на вашому хості.

Розробники, які шукають рішення для якнайшвидшого запуску своїх програм за допомогою Kubernetes з такими функціями, як висока доступність, єдиний вхід і федерація, повинні розглянути пропозиції Kubernetes-як-сервіс, включаючи Platform9. Ці рішення дозволяють зосередитися на розгортанні контейнерних додатків із меншими накладними витратами на оновлення або обслуговування кластера.

Враховуючи масштаб проекту, звичайно, розгортання мікросервісної програми, яка вже розміщена в контейнері в системі оркестровки контейнерів, є цілком розумною і доцільною. Виходячи з проведених у цій роботі досліджень, можна зробити висновок, що загальна схема повинна складатися з наступних етапів:

- Публікація налаштованих контейнерів в репозиторій контейнерів

- Розгортка та конфігурація контейнерів з використанням Kubernetes
- Створення групи кластерів
- Створення кластера в групі
- Налаштування маніфесту розгортки та інші мережеві налаштування для забезпечення доступу
- Розгортка кластеру в середовищі
- Моніторинг роботи та ресурсів кластеру

ВИСНОВКИ

Під час виконання магістерської роботи було досліджено сучасні засоби та методи розробки і розгортки мікросервісних додатків, проаналізовані сучасні інструменти для імплементації DevOps практик, та формалізовано функціональні и технічні вимоги до розроблюваної системи і її розгортки. Основні критерії дослідження – це швидкість налаштування інфраструктури та відповідність потребам підприємства.

На підставі аудиту біло виявлено ряд проблем, які можна вирішити шляхом налаштування систем постійної інтеграції, постійної розгортки, використання статичних аналізаторів та використанням технологій контейнеризації, а саме інструментів Docker та Kubernetes з DevOps практики.

Для хостингу веб-додатків та будь-якої складності найкращім провайдером хмарних обчислень було обрано сервіс AWS. Підтримка кратного масштабування так виключна стікйсть до відмов – це ключові характеристики за якими було зроблено такий висновок.

Мікросервісний підхід повністю виправдовує своє використання кратним спрощенням процесу розробки у розподілених системах шляхом розділу монолітної системи на ряд підсистем що комунікують між собою та складають один великий програмний комплекс.

ПЕРЕЛІК ПОСИЛАНЬ

1. Документація Docker конейнерів [Електронний ресурс] – Режим доступу до ресурсу: https://www.docker.com/

2. Документація Kubernetes оркестартору [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/home/

3. Sam Newman, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, 2019, 111 с.

4. Mark Masseos, Design Rules for RES 2st Edition, 2012. – 115 с.

5. Jason Cannon, Docker:Project-Based Approach to Learning, 2021. – 199 с.

6. Docker Hub [Електронний ресурс] – Режим доступу до ресурсу: https://hub.docker.com/

7. Robin Wieruch, The Road to DevOps: road to mastery 2016. – 228 с.

8. Nigel Poulton, Quick Start Kubernetes 2021. – 118 с.

9. Scott Surovich, Kubernetes and Docker - An Enterprise Guide: Effectively containerize applications 2020. – 366 с.

10. Nigel Poulton, Docker Deep Dive: Zero to Docker in a single book 2016. – 46 с.

11. Russ McKendrick, Mastering Docker: Enhance your containerization and DevOps skills to deliver production-ready applications 2010. – 150 с.

12. James Turnbull, The Docker Book: Containerization is the new virtualization, 2014. – 298 с.

13. Nigel Poulton, The Kubernetes Book: Updated April 2021, 2021. – 116 с.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ДНІПРОВСЬКА ПОЛІТЕХНІКА”

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ
ТЕКСТ ПРОГРАММИ

Текст програми
804.02070743.22015-01 12 01

Листів 21

2022

АНОТАЦІЯ

Даний лістинг містить в собі частину програмного коду для програмування роботи клієнтської частини застосунку. Показані основні підходи до роботи с браузерними сховищам, перехват запитів, маршрутизація запитів на сторінки, робота з історією, робота з компонентом контролю взаємодії з даними про автентифікацію що знаходяться у сховищі, ініціалізація константних значень, та файли конфігурацій.

ЗМІСТ

1. Створення точки входу застосунку 4
2. Налаштування запитів до базового API 5
3. Налаштування перехвату запитів 5
4. Маршрутизація запитів на сторінки 8
5. Робота із локальним сховищем 11
6. Компонент контролю автентифікації 12
7. Ініціалізація базових константних значень 14
8. Налаштування статичного аналізу коду 15
9. Конфігурація метаданих застосунку 16
10. Налаштування маршрутизації діалогових вікон 18
11. Метод обрізки зображень за допомогою canvas API 19
12. Запит на інвалідацію токену 20
13. Конфігурація системи контролю версій 21

1. Створення точки входу застосунку

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <link href="https://fonts.googleapis.com/css?family=Roboto:500,700&display=swap"
rel="stylesheet">
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1,
user-scalable=no">
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Staff Management software"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>Staff Management</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
// ініціалізація головного компоненту сторінки
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import 'semantic-ui-css/semantic.min.css';

ReactDOM.render(
  <React.StrictMode>
```

```
    <App />
  </React.StrictMode>,
  document.getElementById('root'),
);
```

2. Налаштування запитів до базового API

```
import axios from 'axios';
import { responseInterceptor, errorInterceptor, requestInterceptor } from './interceptors';

const baseURL = process.env.REACT_APP_API_URL;

const baseAPI = axios.create({
  baseURL,
  headers: {
    'Content-type': 'application/json; charset=UTF-8',
  },
  withCredentials: true,
});

baseAPI.interceptors.response.use(
  (request) => requestInterceptor(request),
);
baseAPI.interceptors.response.use(
  (response) => responseInterceptor(response),
  (error) => errorInterceptor(error),
);

export default baseAPI;
```

3. Налаштування перехвату запитів

```
import { AUTHORIZATION } from 'api/constants';
import storage from './storage';
import history from '../history';
```

```
const handleSignIn = (response) => {
  const NEW_PASSWORD_REQUIRED = 'New password is required';

  if (response.data?.message === NEW_PASSWORD_REQUIRED) {
    const userEmail = response.config.data && JSON.parse(response.config.data).email;
    storage.set('authorization', AUTHORIZATION.PASSWORD_CHANGE);
    storage.set('email', userEmail);
  } else if (response.data?.userID) {
    storage.set('userId', response.data?.userID);
    storage.set('email', '');
    storage.set('authorization', AUTHORIZATION.AUTHORIZED);
  }

  return response;
};

const handleSignUp = (response) => {
  const isAuthorized = storage.get('authorization') == AUTHORIZATION.AUTHORIZED;

  if (!response.data?.message) {
    if (isAuthorized) {
      history.push('/');
    } else {
      history.push('/login');
    }
  }

  return response;
};

const handleReset = (response) => {
  if (!response.data?.message) {
```

```
    storage.set('authorization', AUTHORIZATION.UNAUTHORIZED);
    history.push('/login');
  }

  return response;
};

const handleUser = (response) => {
  const role = response.data?.role;

  if (role) {
    storage.set('role', role);
  }

  return response;
};

const handleSignOut = () => {
  storage.set('authorization', AUTHORIZATION.UNAUTHORIZED);
  storage.set('role', '');
  storage.set('email', '');
  storage.set('userId', '');
  history.push('/login');
};

export const responseInterceptor = (response) => {
  if (response.status === 401) {
    handleSignOut();
    return response;
  }

  switch (response.config.url) {
    case '/signin': return handleSignIn(response);
```

```
    case '/signup': return handleSignUp(response);
    case '/user': return handleUser(response);
    case '/password/required': return handleReset(response);

    default:
      return response;
  }
};

export const requestInterceptor = (request) => request;

export const errorInterceptor = (error) => {
  const customMessage = error.response?.data.message;

  if (error.response?.status === 401) {
    handleSignOut();
  }

  if (error.response?.status === 500) {
    alert(`Request ${error.response.config.url} failed due to server error, page will be reloaded. If
error persists contact administrator`);
    window.location.reload();
  }

  return Promise.reject(
    customMessage ? new Error(customMessage) : error,
  );
};
```

4. Маршрутизація запитів на сторінки

```
import React, {
  lazy,
  Suspense,
```

```
  useState,
  useEffect,
} from 'react';
import {
  Route,
  Switch,
  Router,
  Redirect,
} from 'react-router-dom';
import storage from './api/storage';
import Layout from './layout';
import SignUp from './pages/signup';
import { AUTHORIZATION } from './api/constants';
import history from './history';

import Home from './pages/home';
import Task from './pages/task';
import Login from './pages/login';
import Reset from './pages/reset';
import SignOut from './pages/sign-out';
import Vacations from './pages/vacations';
/*
const Task = lazy(() => import('./pages/task'));
const Home = lazy(() => import('./pages/home'));
const Login = lazy(() => import('./pages/login'));
const SignOut = lazy(() => import('./pages/sign-out'));
const Reset = lazy(() => import('./pages/reset'));
*/

const App = () => {
  const [authorization, setAuthorization] = useState(storage.get('authorization'));

  useEffect(() => {
```

```
  const authListener = () => {
    const auth = storage.get('authorization');

    if (Object.values(AUTHORIZATION).includes(auth)) {
      setAuthorization(auth);
    } else {
      storage.set('authorization', AUTHORIZATION.UNAUTHORIZED);
      setAuthorization(AUTHORIZATION.UNAUTHORIZED);
    }
  };

  authListener();

  return storage.subscribe(authListener);
}, []);

return (
  <Router history={history}>
    <Layout>
      <Suspense fallback={<div>Loading...</div>}>
        {
          authorization === AUTHORIZATION.AUTHORIZED && (
          <Switch>
            <Route path="/" exact component={Home} />
            <Route path="/create" component={Home} />
            <Route path="/user/:id" exact component={Home} />
            <Route path="/task/:id" component={Task} />
            <Route path="/vacations/:id" component={Vacations} />
            <Route path="/vacations/" component={Vacations} />
            <Route path="/adduser" exact component={SignUp} />
            <Route path="/signout" exact component={SignOut} />
            <Route path="/reset" exact component={Reset} />
            <Redirect to="/" />
```

```
          </Switch>
          )
        }
        {
          authorization === AUTHORIZATION.PASSWORD_CHANGE && (
          <Switch>
            <Route path="/reset" exact component={Reset} />
            <Redirect to="/reset" />
          </Switch>
          )
        }
        {
          authorization === AUTHORIZATION.UNAUTHORIZED && (
          <Switch>
            <Route path="/adduser" exact component={SignUp} />
            <Route path="/login" exact component={Login} />
            <Redirect to="/login" />
          </Switch>
          )
        }
      </Suspense>
    </Layout>
  </Router>
 );
};

export default App;
```

5. Робота із локальним сховищем

```
const storage = {
  get(key, defaultValue) {
    const value = window.localStorage.getItem(key);
```

```
    return value === null ? defaultValue : value;
  },
  set(key, value) {
    try {
      window.localStorage.setItem(key, value);
      window.dispatchEvent(new Event('storage:change'));
    } catch (error) {
      console.error('Can\'t set item to storage:', key, value);
    }
  },
  remove(key) {
    window.localStorage.removeItem(key);
    window.dispatchEvent(new Event('storage:change'));
  },
  subscribe(listener) {
    window.addEventListener('storage:change', listener);

    return () => window.removeEventListener('storage:change', listener);
  },
};

export default storage;
```

6. Компонент контролю автентифікації

```
import React, { useEffect, useState } from 'react';
import { AUTHORIZATION, USER_ROLE } from 'api/constants';
import storage from 'api/storage';

const withAuth = (WrappedComponent) => (props) => {
  const [userId, setUserId] = useState(storage.get('userId'));
  const [role, setRole] = useState(storage.get('role'));
  const [isAuthorized, setIsAuthorized] = useState(storage.get('authorization') ===
AUTHORIZATION.AUTHORIZED);
```

```
useEffect(() => {
  const authListener = () => {
    const storedAuth = storage.get('authorization');
    const storedRole = storage.get('role');
    const storedUserId = storage.get('userId');

    setUserId(storedUserId);

    if (Object.values(USER_ROLE).includes(storedRole)) {
      setRole(storedRole);
    }

    if (storedAuth === AUTHORIZATION.AUTHORIZED) {
      setIsAuthorized(true);
    } else {
      setIsAuthorized(false);
    }
  };

  authListener();

  return storage.subscribe(authListener);
}, []);

return (
  <WrappedComponent
    // eslint-disable-next-line react/jsx-props-no-spreading
    {...props}
    authorizedUserId={userId}
    userRole={role}
    isAdmin={role === USER_ROLE.ADMIN}
    isAuthorized={isAuthorized} />
```

```
  );
};

export default withAuth;
```

7. Ініціалізація базових константних значень

```
import { USER_ROLE } from 'api/constants';

export const ROLES = [
  { key: USER_ROLE.USER, value: USER_ROLE.USER, text: 'User' },
  { key: USER_ROLE.ADMIN, value: USER_ROLE.ADMIN, text: 'Manager' },
];

export const TASK_STATE = {
  IN_PROGRESS: 'InProgress',
  READY_FOR_DEV: 'Ready',
  BLOCKED: 'Blocked',
  DONE: 'Done',
};

export const TASK_STATE_STRING = {
  InProgress: 'In progress',
  Ready: 'Ready for dev',
  Blocked: 'Blocked',
  Done: 'Done',
};

export const TASK_STATE_OPTIONS = [
  { key: TASK_STATE.IN_PROGRESS, value: TASK_STATE.IN_PROGRESS, text:
TASK_STATE_STRING[TASK_STATE.IN_PROGRESS] },
  { key: TASK_STATE.READY_FOR_DEV, value: TASK_STATE.READY_FOR_DEV, text:
TASK_STATE_STRING[TASK_STATE.READY_FOR_DEV] },
```

```
  { key: TASK_STATE.DONE, value: TASK_STATE.DONE, text:
TASK_STATE_STRING[TASK_STATE.DONE] },
  { key: TASK_STATE.BLOCKED, value: TASK_STATE.BLOCKED, text:
TASK_STATE_STRING[TASK_STATE.BLOCKED] },
];

export const VACATION = {
  PENDING: 'Pending',
  APPROVED: 'Approved',
  REJECTED: 'Rejected',
  CANCELED: 'Canceled',
  EXPIRED: 'Expired',
};

export const USER_ROLE = {
  ADMIN: 'admin',
  USER: 'user',
};

export const AUTHORIZATION = {
  AUTHORIZED: 'AUTHORIZED',
  PASSWORD_CHANGE: 'PASSWORD_CHANGE',
  UNAUTHORIZED: 'UNAUTHORIZED',
};
```

8. Налаштування статичного аналізу коду

```
{
  "settings": {
    "import/resolver": {
      "node": {
        "paths": ["src"]
      }
    }
```

```
  },
  "extends": [
    "react-app",
    "airbnb",
    "plugin:jsx-a11y/recommended"
  ],
  "plugins": [
    "jsx-a11y"
  ],
  "rules": {
    "react/jsx-filename-extension": [1, { "extensions": [".js", ".jsx"] }],
    "react/forbid-prop-types": "off",
    "react/destructuring-assignment": "off",
    "react/state-in-constructor": "off",
    "jsx-a11y/click-events-have-key-events": "off",
    "jsx-a11y/no-static-element-interactions": "off",
    "jsx-a11y/label-has-associated-control": "off",
    "import/prefer-default-export": "off",
    "react/jsx-closing-bracket-location": [1, "after-props"],
    "max-len": ["error", { "code": 120 }]
  }
}
```

9. Конфігурація метаданих застосунку

```
{
  "name": "STFMGMT",
  "version": "1.0.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "axios": "^0.19.2",
```

```
  "classnames": "^2.2.6",
  "debounce": "^1.2.0",
  "husky": "^4.2.5",
  "moment": "^2.25.3",
  "prop-types": "^15.7.2",
  "pure-react-carousel": "^1.27.0",
  "react": "^16.13.1",
  "react-dates": "^21.8.0",
  "react-dom": "^16.13.1",
  "react-dropzone": "^11.0.1",
  "react-image-crop": "^8.6.2",
  "react-input-autosize": "^2.2.2",
  "react-router-dom": "^5.1.2",
  "react-scripts": "3.4.1",
  "semantic-ui-css": "^2.4.1",
  "semantic-ui-react": "^0.88.2"
},
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "lint": "eslint ."
},
"browserslist": {
  "production": [
    ">0.2%",
    "not dead",
    "not op_mini all"
  ],
  "development": [
    "last 1 chrome version",
    "last 1 firefox version",
```

```
    "last 1 safari version"
   ]
  },
  "devDependencies": {
   "eslint-config-airbnb": "^18.1.0",
   "eslint-config-prettier": "^6.10.1",
   "eslint-plugin-jsx-a11y": "^6.2.3",
   "eslint-plugin-prettier": "^3.1.3"
  },
  "husky": {
   "hooks": {
    "pre-commit": "yarn build"
   }
  }
}
```

10. Налаштування маршрутизації діалогових вікон

```
import React from 'react';
import {
  Switch,
  Route,
} from 'react-router-dom';
import TaskCreateModal from 'components/task-create-modal';
import VacationRequestModal from 'components/vacation-request-modal';
import history from '../../history';

const Modals = () => {
  const closeTaskModal = () => {
    history.push('/');
  };

  return (
    <Switch>
```

```
    <Route path="/create/task" component={() => <TaskCreateModal
onClose={closeTaskModal} />} />
    <Route path="/create/vacation" component={() => <VacationRequestModal
onClose={closeTaskModal} />} />
   </Switch>
  );
};

export default Modals;
```

11. Метод обрізки зображень за допомогою canvas API

```
const getCroppedImg = (image, crop, blob = false) => {
  if (!crop.width) return null;
  const canvas = document.createElement('canvas');
  const scaleX = image.naturalWidth / image.width;
  const scaleY = image.naturalHeight / image.height;
  canvas.width = crop.width;
  canvas.height = crop.height;
  const ctx = canvas.getContext('2d');

  ctx.drawImage(
    image,
    crop.x * scaleX,
    crop.y * scaleY,
    crop.width * scaleX,
    crop.height * scaleY,
    0,
    0,
    crop.width,
    crop.height,
  );

  if (blob) {
```

```
    const getCanvasBlob = () => new Promise((resolve) => {
      canvas.toBlob((canvasBlob) => resolve(canvasBlob));
    });

    return getCanvasBlob();
  }

  return canvas.toDataURL('image/jpeg', 1);
};

export default getCroppedImg;
```

12. Запит на інвалідацію токену

```
import React, { useEffect } from 'react';
import storage from 'api/storage';
import baseAPI from 'api/config';
import { AUTHORIZATION } from 'api/constants';

const SignOut = () => {
  useEffect(() => {
    const signOut = async () => {
      await baseAPI.post('/signout');
      storage.set('authorization', AUTHORIZATION.UNAUTHORIZED);
      storage.set('role', '');
      storage.set('email', '');
      storage.set('userId', '');
    };
    signOut();
  }, []);

  return 'Signing out';
};
```

```
export default SignOut;
```

13. Конфігурація системи контролю версій

```
# dependencies
/node_modules
/.pnp
.pnp.js

# testing
/coverage

# production
/build

# misc
.DS_Store
.idea
.env
.env.local
.env.development.local
.env.test.local
.env.production.local

npm-debug.log*
yarn-debug.log*
yarn-error.log*
```